AFRL-IF-RS-TR-2000-103
Final Technical Report
July 2000

SECURE DISTRIBUTED TRANSACTION PROCESSING

SRI International

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D337

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2000-103 has been reviewed and is approved for publication.

APPROVED: Michael P. Nassif

MICHAEL P. NASSIF
Project Engineer

FOR THE DIRECTOR: W. Debany

WARREN H. DEBANY, JR, Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGB, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

SECURE DISTRIBUTED TRANSACTION PROCESSING

R. A. Riemenschneider

Contractor: SRI International
Contract Number: F30602-95-C-0277
Effective Date of Contract: 15 July 1995
Contract Expiration Date: 15 July 1998

Short Title of Work: Secure Distributed Transaction Processing
Period of Work Covered: Jul 95 – Jul 98

Principal Investigator: R. A. Riemenschneider
Phone: (650) 859-2000
AFRL Project Engineer: Michael P. Nassif
Phone: (315) 330-3342

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Michael P. Nassif, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	JULY 2000	Final Jul 95 - Jul 98

4. TITLE AND SUBTITLE

SECURE DISTRIBUTED TRANSACTION PROCESSING

5. FUNDING NUMBERS

C - F30602-95-C-0277
PE - 62301E
PR - C929
TA - 02
WU - 06

6. AUTHOR(S)

R.A. Riemenschneider

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

SRI International
333 Ravenswood Avenue
Menlo Park CA 94025-3493

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2000-103

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Michael P. Nassif/IFGB/(315) 330-3342

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This technical report describes the work of the Dependable System Architecture Group on the Secure Distributed Transaction Processing (SDTP) project. Chapter 1 provides an overview of secure software architectures. Chapter 2 describes the goals and methodology of the SDTP project. Chapter 3 of this report provides a proof of the model-theoretic approach used in the project to prove that refinement steps are faithful and preserve both security and functional properties. Chapter 4 introduces an alternative method of proving refinement. Chapter 5 describes the method for showing patterns are faithful and hence security-preserving. Chapter 6 shows refinement patterns that do not always preserve a property of interest, such as security, can be used without losing the correctness guarantee that a restriction to validated refinement patterns automatically provides. Chapter 7 discusses a case study in architecture verification. Finally, Chapter 8 provides more detail on the reference implementation and describes two applications of the reference implementation (law enforcement and intrusion detection).

14. SUBJECT TERMS

Information Security, Computer Security, Transaction Processing

15. NUMBER OF PAGES

140

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

Contents

Chapter 1

Introduction

The following chapters describe the work of the Dependable System Architecture group of SRI International's System Development Laboratory[1] on the Secure Distributed Transaction Processing (SDTP) project. Each chapter was originally a technical report or article. They have been lightly edited — for example, notational conventions are much more consistent than in the originals — but the redundancy that allows each to be read independently of the others has been retained. You can read only those chapters that interest you, and skip the rest.

Chapter 2 was originally published in the proceedings of the 1997 IEEE Symposium on Security and Privacy, held in Oakland, CA [27]. The authors of this chapter are R. A. Riemenschneider, Mark Moriconi (now at SecureSoft), Xiaolei Qian (also at SecureSoft), and Li Gong (now at JavaSoft). It describes the goals and methodology of the project, and explains why achieving those goals would be significant. The basic idea of bridging the gap between theory and practice by linking a high-level, demonstrably secure system model to a reference implementation by refinement steps that preserve security generated considerable interest.

Chapter 3 appeared as a technical report [38] by R. A. Riemenschneider. It provides a proof of the model-theoretic result used in Chapter 2 to prove refinement steps are "faithful", and thus preserve security as well as functional properties. It seems unlikely that this result is original. But, to the best of our knowledge, a proof has never appeared in print. Given the central importance of the result to our method of establishing correctness, we wanted to make it widely available.

Chapter 4 also appeared as a technical report [39] by R. A. Riemenschneider. It introduces an alternative method of proving refinement steps are faithful that is much simpler to apply, but is less widely applicable than the original. It also contains a detailed proof that one of our standard refinement examples, replacement of a dataflow channel between two functions by a shared variable

[1] The DSL group was part of the Computer Science Laboratory prior to the founding of the System Development Laboratory this year.

that is written by one and read by the other, faithfully interprets the theory of the dataflow structure in the theory of the shared variable structure, using the simplified method.

Chapter 5 is the last in the series of technical reports by R. A. Riemenschneider describing the method for showing refinement *patterns* are faithful, and hence security-preserving [41]. It is primarily devoted to arguing that our previous proofs established only the faithfulness of particular refinement steps (i.e., instances of refinement patterns), and not the faithfulness of the generally applicable pattern, because crucial contextual features are ignored. It is demonstrated that faithfulness of the refinement step is not sufficient to guarantee faithfulness of the general pattern. But a technique for converting proofs of "contextless" refinement steps into proofs of refinement transformations that can be applied in *most* contexts is presented.

Chapter 6, also by R. A. Riemenschneider, shows how refinement patterns that do not always preserve a property of interest, such as security, can be used without losing the correctness guarantee that a restriction to validated refinement patterns automatically provides. This flexibility was crucial to our strategy of basing the SDTP derivation on our (non-secure) X/Open DTP derivation. (This strategy is explained in more detail in Chapter 7.) We had established that the particular refinement steps in the DTP derivation were faithful, but not that the general patterns employed were always faithful. The technique developed in Chapter 5 usually guaranteed faithfulness of the corresponding step in the SDTP derivation, but occasionally failed to do so because contextual restrictions were violated. The basic idea behind checking refinement steps, inspired by recent work on proof-carrying code, is to refine a proof of, say, architectural security properties in parallel with the refinement of the architecture itself. If the refined proof of the security of the more abstract architecture can be turned into a proof of the security of the refined architecture, then the refined architecture clearly has the desired security property. This chapter originally appeared in the IFIP volume on Software Architecture [40]. As it was written for a general software engineering audience, it makes fewer demands on the reader's mathematical skills than the previous three chapters.

Chapter 7, by Fred Gilham, R. A. Riemenschneider, and Victoria Stavridou, was presented at the recent World Congress on Formal Methods (FM '99) as a case study in architecture verification. It describes the derivation of the SDTP reference implementation from the abstract, "secure dataflow style" description of the SDTP architecture, and provides a nice summary of the results of the project. A description of the first application of the reference implementation — coordinating access to distributed, multilevel law enforcement databases — is also included.

Finally, Chapter 8, a paper written for the upcoming Lisp User Group Meeting by Fred Gilham and David Shih, provides more detail on the reference implementation. It also describes the two applications of the reference implementation, the law enforcement application and a recent application that accesses distributed, multilevel intrusion detection databases, with the object of finding coordinated patterns of attack across sites.

Chapter 2

Secure Software Architectures

2.1 Chapter Overview

In recent years, there has been a growing demand for vendor-neutral, open systems solutions. These solutions take the form of "open software architectures" that represent a family of systems. Open architectures are critical tools for competitive success in the information technology industry [29]. They enable vendors to substantially reduce time-to-market and development costs, since they do not have to provide an integrated solution in the context of a bewildering array of graphical user interfaces, operating systems, and connectivity schemes. Moreover, open architectures enable consumers to have confidence that future purchases will easily integrate with existing systems.

Several consortia have been created to develop open software architectures. For example, the Object Management Group (OMG), which consists of over 600 companies, has developed a widely accepted architecture, called the Common Object Request Broker (CORBA), that allows applications to communicate seamlessly in a heterogeneous, distributed environment. Interoperation architectures also have been developed by Sun Microsystems (Tooltalk), Xerox PARC (ILU), Open Software Foundation (DCE), and Microsoft (OLE), among others.

An example of an important open application architecture is the X/Open Distributed Transaction Processing (DTP) reference architecture. X/Open DTP is intended to standardize the interactions and communications between the components of the 3-tiered client/server model. Figure 2.1 illustrates a representative 3-tiered model in which presentation aspects are handled by "thin" clients, business logic is incorporated in the transaction processing monitor (e.g., BEA's Tuxedo™), and data/resource management is the responsibility of data servers (e.g., Oracle 7, IBM/DB2, and Sybase 11).

The X/Open DTP reference architecture allows multiple application programs to share heterogeneous resources provided by multiple resource managers,

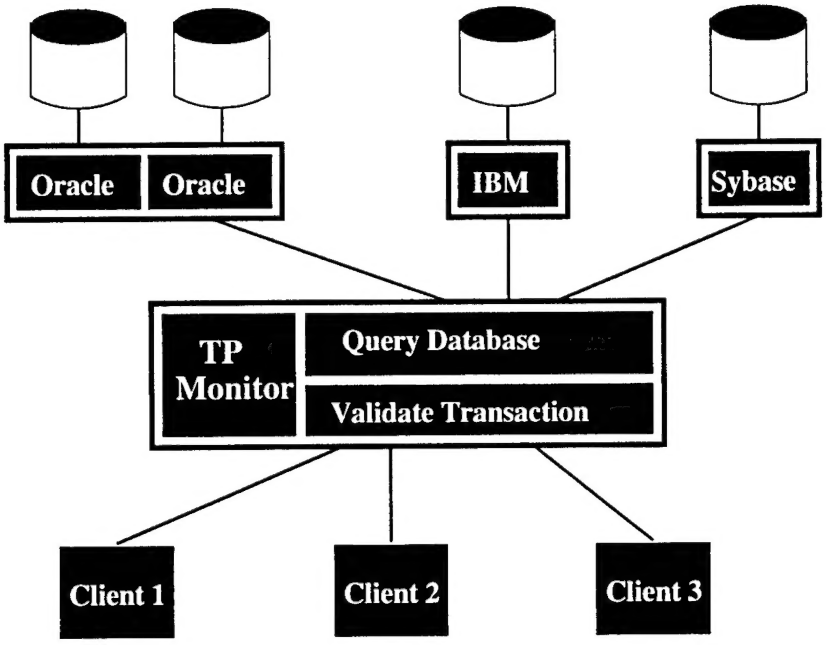

Figure 2.1: 3-Tiered Client-Server Architecture

and allows their work to be coordinated into global transactions. If the X/Open DTP interfaces (APIs) are adhered to, the components of a particular DTP system (namely, every application, transaction manager, resource manager, and resource) will be portable, interchangeable, and interoperable.

Tremendous leverage can be gained by incorporating security directly into architectural standards. This opportunity has been recognized, for example, by OMG in its attempts to extend CORBA for secure object access and communication. An extension of X/Open DTP for secure transaction processing would enable vendors to develop single- or multilevel products with little or no concern about the security of the overall transaction processing system in which they will be used. Furthermore, such an extension would allow users and application vendors to concentrate on the selection of components, knowing with confidence that the overall DTP system will be secure.

The benefits of a secure architectural standard can be realized only if there is high and justifiable confidence that any instance of it satisfies the intended security property. An erroneous assumption that every instance of an X/Open DTP architecture properly controls access to protected information could have serious legal and business consequences. In this paper, we propose a formal approach to secure architectures that involves three steps:

1. Formalization of the system architecture in terms of common architectural abstractions.

2. Refinement of the system architecture into specialized architectures, each suitable for implementation under different assumptions about the security of the system components.[1]

3. Rigorous proof that every implementation that conforms to the system architecture, or one of its specializations, satisfies the intended security policy.

Our approach is made difficult by the dominance of informal architectures[2] and security theories that are difficult to connect to implementations. Fortunately, we can overcome these difficulties by combining recent results from the software research community [10, 12, 19, 25, 26] with well-known results from the security community.

To illustrate our approach, we present excerpts from our formalization of a secure version of the X/Open DTP standard — called SDTP. It consists mostly of structural information involving component interfaces and the connections among them. It also contains causal dependencies between certain interfaces and connections, but they are not required to demonstrate secure access control. The access control property was chosen because it is well understood and because it is important in the commercial and the military sectors.

[1] Refinement also can be used to accommodate different computing and networking environments, but that is not the focus of this paper.

[2] The X/Open DTP standard consists of approximately 500 pages of diagrams, C interfaces, and English explanations.

The development of a particular secure architecture should take into account two important observations, both of which are illustrated in the subsequent discussion of SDTP.

1. The interplay between architectural and security concerns can dramatically affect the practicality of a solution. For example, the obvious way to add security to X/Open DTP has the consequence that vendors must build multilevel components — even though they are not security experts and may want to target larger, single-level markets.

2. Rather than modeling a system as a single architecture, it should be modeled as multiple, but related, architectures — each making different assumptions about the security of the system components and the protocols.

2.2 Related Work in Security

There are two main challenges in developing secure (software or hardware) systems. One is to obtain a thorough understanding of the desired security properties and the relevant (functional and non-functional) properties of the target system. This normally calls for a formal model of the system together with security proofs, often given in some mathematical or logical language. The second challenge is to assure that the actual system implementation realizes the more abstract formal model.

The research community has spent considerable effort on the first of these challenges. The security properties considered include non-interference, information flow, and composability, with system models built using traces, CSP, and other formal languages [13, 21, 22, 23]. These behavioral models are far removed from the actual systems, making it extremely hard, if not impossible, to be convinced that an implementation satisfies the security properties proven about the model.

On the other hand, commercially available systems, such as OSF's DCE 1.1, include a wide range of security functionalities, such as authentication and access control. But these architectures cannot be linked formally to a solid theory of security and, given the overall complexity of these architectures, it is difficult to distill whether or not they provide the desired security properties. The same observation applies to uses of the "reference monitor" concept that was popular in the 1980s. Systems designed around this concept could not reliably be connected to implementations.

A middle ground between abstract mathematical modeling and system-level analysis can be seen in connection with the Rampart system [37]. Rampart is a distributed system for which an attempt was made to prove that it satisfies certain security and fault-tolerance properties. However, the proof is with regard to an abstract algorithmic model, which again suffers from the difficulty in relating it to the actual implementation.

A successful effort to connect security requirements for a secure gateway to its implementation is described in [33, 34]. A convincing, but not formal,

argument is made that the connection preserves security. The argument depends on the use of architectural elements that are very similar to those used in the implementation. Our architecture descriptions [25, 26, 28] are more general in three important ways. First, an architecture hierarchy can include both horizontal and vertical decomposition (change in representation), whereas the gateway development involves only horizontal. Second, formal mappings between architectures are used to bridge the gap in vertical hierarchies. Third, our architecture definition language makes it easy to define generic architectures. These three capabilities are useful in defining practical architecture standards, such as X/Open DTP, and are useful in architecting any large system. By formalizing mappings and correctness arguments, we can prove that lower levels in a vertical hierarchy — ultimately, the implementation — preserve security properties proved at higher levels.

2.3 The X/Open DTP Standard

Distributed transaction processing captures the core activities in distributed information systems, namely, the interaction between applications and resources. Transactions form the basis of such interaction, which are multiple application programs running concurrently and accessing shared data resources. Distributed transaction processing is ubiquitous in military and commercial applications, many of which have stringent security requirements.

The X/Open DTP standard architecture is described in a series of publications of X/Open Ltd. [47, 49, 48, 46]. An executable prototype of the DTP standard appears in Luckham et al. [19]. The standard describes a particular set of component interfaces, and sequences of interactions between the components. Components may be various application programs, resource managers (such as databases, file systems, or mailers), and transaction managers. The purpose is to define a standard communication architecture through which multiple application programs may share resources concurrently by organizing their activities into transactions which appear to be atomic. Transactions, which may execute concurrently, can contain many operations on resources. *Atomicity* means that after a transaction executes, either all or none of its operations take effect.

The X/Open description is informal, consisting of English text together with interfaces given in the C programming language. The important features of the standard are (1) the interfaces, (2) the architecture (the ways of connecting components that satisfy the standard), and (3) the protocols (calling sequences) for using the interfaces. The calling sequences are described in terms of a single thread of control and C function calls. Many different systems with various applications and resources may satisfy this standard. Those that do should be easier to combine, thus promoting the goal of "open" systems.

A version of the X/Open architecture, shown in Figure 2.2, consists of three types of components — one application program (AP), one transaction manager (TM), and one or more resource managers (RMs). The boxes indicate the component interfaces, and the lines indicate the communication between them.

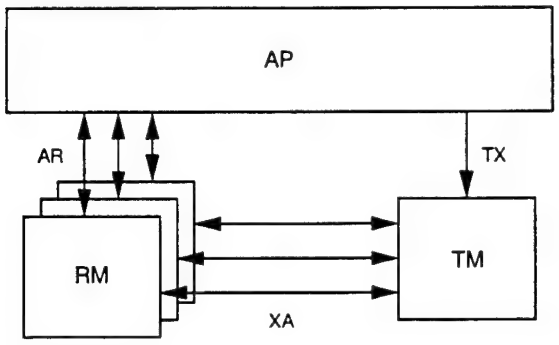

Figure 2.2: X/Open DTP Reference Architecture

The label TX indicates a complex connection and protocol defining communication between any application module and any transaction manager. The TX connection contains connections between functions for initiating and finalizing transactions. Communication is always initiated by the application. A series of calls back and forth continues until communication is completed. Similar complex connections exist between the application and every resource (the AR connection), and between the transaction manager and every resource manager (the XA connection). The XA connection involves the well-known two-phase commit protocol for ensuring atomicity. Much of this activity can be concurrent, and many transactions may take place at once.

2.4 Formal Model of X/Open DTP

We first show how to define the X/Open DTP interfaces, components, and wiring using an architecture definition language called SADL [28].[3] Then, we describe how architectures written in SADL can be translated systematically into logic. All the proofs in this paper can be fully formalized in terms of the logical theories that result from the translation.

An interface is defined as a type so that it can be associated with more than one component. For example, we define the interface for transaction managers as follows.

```
tm: TYPE <=
  MODULE
  EXPORTING ALL
  BEGIN
    register: AX_Register_Procedure
      [id: X_Id, rmid: INT, flags: INT
        -> ret: INT]
```

[3] SADL is an acronym for Structural Architecture Definition Language.

```
  unregister: AX_Unregister_Procedure
    [rmid: INT, flags: INT
       -> ret: INT]
  begin: TX_Begin_Procedure
    [ -> ret: INT]
  close: TX_Close_Procedure
    [ -> ret: INT]
  commit: TX_Commit_Procedure
    [ -> ret: INT]
  information: TX_Info_Procedure
    [info: TX_Info -> ret: INT]
  open: TX_Open_Procedure
    [ -> ret: INT]
  rollback: TX_Rollback_Procedure
    [ -> ret: INT]
END
```

This declaration says that every object of type `tm` is a module containing a definition of the procedures `register`, `unregister`, `begin`, and so on, used in the X/Open specification of the TX interface. For example, `begin` is declared to be a `TX_Begin_Procedure`, which guarantees that it has an appropriate "semantics" relative to its role in the protocol. `TX_Begin_Procedure` is declared elsewhere to be a subtype of `Procedure`.

As an example of a more complex interface, consider the definition of the application interfaces, which must allow for an arbitrary number of resources.

```
ap: TYPE <=
   MODULE
      [<< ap_in1(i): r_type(i)
                  |(i: NAT) i < n >>
         -> << ap_out1(i): q_type(i)
                  |(i: NAT) i < n >>]
```

This declaration says that an object of type `ap` is a module with `n` input ports (`n` is the number of resource managers in the system being specified) and `n` output ports. The type of the `i`-th input port, `ap_in1(i)`, is `r_type(i)`, where `r_type` has been declared to be a function from non-negative integers less than `n` to subtypes of the type `ar_resources` of data sent from the RMs to the AP. Similarly, the type of the `i`-th output port, `ap_out1(i)`, is `q_type(i)`, where `q_type` has been declared to be a function from non-negative integers less than `n` to subtypes of the type `ar_requests` of data sent from the AP to the RMs.

The declaration of a resource manager, denoted by the `rm` type, contains procedure calls, as in the `tm` declaration, and ports, as in the `ap` declaration. Since an instance of DTP can contain an arbitrary number of resources, the type `rms` is used to denote the union of an arbitrary number of resource managers.

To define a component, we simply declare an instance of an interface type. The declarations

```
the_ap:  ap
the_rms: rms
the_tm:  tm
```

define the three components in X/Open DTP.

The interfaces of these components are wired together by constraints associated with the AR, XA, and TX connections. Here is a representative constraint on the TX connection.

```
tx_1: ASSERTION =
  Called_From(the_tm.begin, the_ap)
```

says that the procedure `begin` of the TX protocol declared in `the_tm` is called by `the_ap`. The XA interface definition consists of twelve assertions that are similar, except that they are general so as to apply to all RMs. An example is

```
xa_1: ASSERTION =
  (FORALL y: rm)
    Called_From(the_tm.register, y)
```

The AR interface specification is more abstract, since it is described in terms of a general dataflow connection which need not be implemented as a procedure call. The assertion

```
ar_1: ASSERTION =
  (FORALL i: NAT | i < n)
   (EXISTS c: Channel[q_type(i)])
     Connects(c, the_ap.ap_out1(i),
       the_rms.rm_in1(i))
```

says that, for every RM index `i`, there is a dataflow channel `c` which carries the appropriate subtype of `ar_requests` from the `i`-th output port of `the_ap` to the input port of the `i`-th RM in the collection of all RMs, `the_rms`. A similar assertion characterizes the flow from the RMs to the AP.

SADL specifications can be translated systematically into logic, allowing us to regard architectures as logical theories. As an illustration, consider the declaration

```
begin: TX_Begin_Procedure
  [ -> ret: INT ]
```

that appears in the `tm` type. It is translated into

$$\forall x\,[\,\mathsf{TM}(x) \rightarrow \exists y\,\mathsf{TX\_Begin\_Procedure}(y,x)\,]$$

$$\forall x\,\forall y\,[\,\mathsf{TM}(x) \wedge \mathsf{TX\_Begin\_Procedure}(y,x) \rightarrow \exists z\,\mathsf{Return\_Parameter}(z,y)\,]$$

$$\forall x\,\forall y\,\forall z\,\forall w\,[\,\mathsf{TM}(x) \wedge \mathsf{TX\_Begin\_Procedure}(y,x) \wedge \mathsf{Return\_Parameter}(z,y) \wedge \mathsf{Integer}(w) \rightarrow \mathsf{May\_Hold\_Value\_On\_Return}(z,w)\,]$$

This says that every transaction manager has a begin procedure, that every begin procedure has a return parameter, and that the value of a begin procedure's return parameter can be an integer.

Among the assertions that define the TX interface is

```
Called_From(the_tm.begin, the_ap)
```

which can be translated to

$$\forall x\,[\,\mathsf{TX\_Begin\_Procedure}(x,\mathsf{the\_tm}) \rightarrow \exists y\,[\,\mathsf{Call\_Site}(y,x) \wedge \mathsf{Location}(y,\mathsf{the\_ap})\,]\,]$$

This says that every TX begin procedure is called from a site located in the application.

Translation to logic is essential for complicated arguments, such as the relative correctness proofs of Section 2.6, and when fully formalized arguments are required. But often, simpler results can be directly established by reasoning in terms of SADL specifications. The next section, which proves the security of alternative DTP architectures, illustrates this kind of rigorous informal reasoning.

2.5 Secure DTP Architectures

Suppose that we want to enforce the multilevel security (MLS) policy in the DTP architecture. A standard model of the MLS policy is the Bell-LaPadula model [16]. Given a set of subjects each with an attached clearance level, and a set of objects each with an attached classification level, the model ensures that information does not flow downward in a security lattice by imposing the following requirements.

- **The Simple Security Property.** A subject is allowed a read access to an object only if the former's clearance level is identical to or higher than the latter's classification level in the lattice.

- **The $\star$-Property**. A subject is allowed a write access to an object only if the former's clearance level is identical to or lower than the latter's classification level in the lattice.

In terms of the DTP, a subject might be a user invoking an application or a transaction manager, and an object could be any data item in a resource. We say a component in DTP is MLS, if input and output of that component are properly labeled with security levels, and the component enforces the MLS policy internally. Similarly, the DTP architecture is MLS, if it enforces the MLS policy.[4]

Notice that the MLS policy regulates the communication between the applications and the resources, which is application-specific and not part of the DTP standard. It is not obvious how the MLS policy can be enforced while still maintaining plug-and-play.

A naive approach is illustrated in Figure 2.3, where each (non-MLS) application or resource is wrapped by an MLS wrapper, which together enforce the MLS policy at the interface. This approach suffers from at least three problems.

- High-assurance technology does not yet exist that allows a non-MLS component (application or resource) to be wrapped to enforce the MLS policy, especially when the component contains untrusted code.

- Even if an MLS wrapper technology did exist, the enforcement of the MLS policy by a wrapper is most likely dependent on the application-specific interface of the underlying component, which destroys the plug-and-play benefit of a standard architecture.

- The alternative is to require that vendors provide MLS components, which would have significant drawbacks: reduced time-to-market, stiff performance penalties, and increased development costs.

Given the fact that most of the COTS and legacy components available today are not MLS, our strategy is therefore to develop not one but several secure DTP standards, each geared toward a particular configuration of applications and resources. In the remainder of this section, we consider three possible configurations.

The simplest configuration is when the application and all the resources are single-level and are at the same level, as shown in Figure 2.4. The DTP standard does not need any extension to enforce the MAC policy, since there cannot be any downward (or cross-level) information flow in the security lattice.

A second possible configuration is when the application and all the resources are single-level but perhaps at different levels, as shown in Figure 2.5. To enforce the MAC policy, the AP and the RMs cannot directly communicate with each other. Instead, communication has to be filtered by an MLS filter, which regulates every access by the AP to the RMs to ensure the simple security

[4]We concern ourselves only with the two properties of the Bell-LaPadula model, not with issues such as object reuse or covert channels.

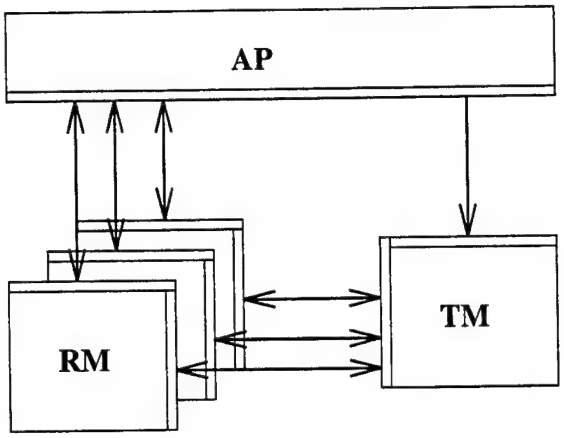

Figure 2.3: Secure DTP with MLS Wrappers

property and the $\star$-property of the Bell-LaPadula model. The TM is required to be MLS since it interacts with components at different levels.

For this scenario, the DTP standard needs to be extended as follows. The begin and register procedure calls in the TM interface are extended with a label parameter to indicate the level of the RM or AP caller.

```
tm: TYPE <=
  MODULE
  EXPORTING ALL
  BEGIN
    register: AX_Register_Procedure
      [id: X_Id, rmid: INT,
            flags: INT,
            lb: LABEL
         -> ret: INT]
    begin: TX_Begin_Procedure
      [lb: LABEL -> ret: INT]
    ...
  END
```

The security of the resulting DTP architecture with respect to the MLS policy can be shown as follows. Suppose that the AP is low and a particular RM r is high. Let us consider the simple security property of the Bell-LaPadula model. The only way that AP can read data in r is through either the TM or the filter. Since communication is secure, any data from r to the TM/filter will be properly labeled high. Since the TM/filter is MLS, it cannot pass high data to the low AP, meaning that AP cannot read data from r. Similarly we can argue that the $\star$-property of the Bell-LaPadula model holds.

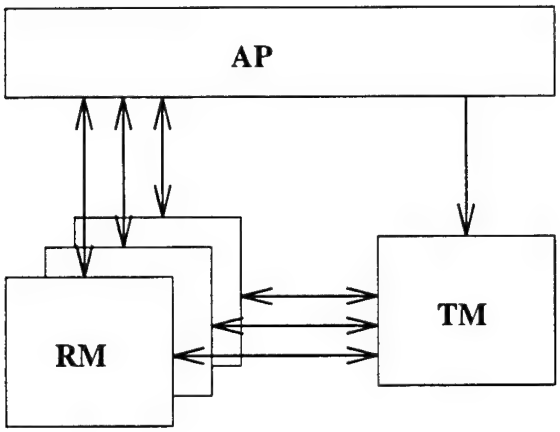

Figure 2.4: Secure DTP with Same-Level AP and RMs

The most complicated configuration, shown in Figure 2.6, is when the application and the resources are multilevel, in contrast to the wrapped components in Figure 2.3. The AP and the RMs can directly communicate with each other, since they are capable of handling data at different levels. Again, the TM should be MLS as well. Conceptually every multilevel connection can be viewed as consisting of multiple single-level connections, one for each level.

The DTP standard needs to be extended as follows. Every AP and RM interface is extended with a label-range parameter to indicate the range of levels of the AP and RM, respectively. Like before, the register and begin procedure calls in the TM interface are extended with a label parameter to indicate the level of the RM or AP caller. In addition, there is a constraint requiring that there is a register procedure call for every level in every RM's range. A similar constraint is needed for the begin procedure call.

```
rm: TYPE <=
  MODULE
  EXPORTING ALL
  BEGIN
    low:  LABEL
    high: LABEL
    label_order: ASSERTION =
      low <= high
    ...
  END

tm: TYPE <=
  MODULE
  EXPORTING ALL
  BEGIN
```

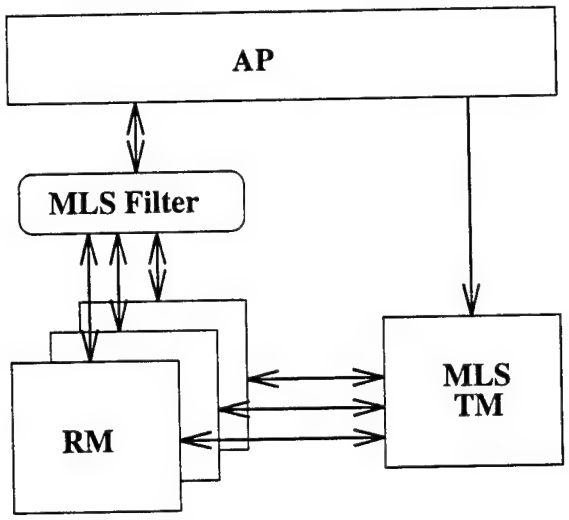

Figure 2.5: Secure DTP with Single-Level AP and RMs

```
    register(lb: LABEL):
      AX_Register_Procedure
        [id: X_Id, rmid: INT,
              flags: INT
           -> ret: INT]
    ...
  END

the_tm: tm

rms_label: ASSERTION =
  (FORALL x: rm)(FORALL y: LABEL)
    [x.low <= y AND y <= x.high
       => Called_From
            (the_tm.register(y), x)]
```

It is straightforward to prove the security of this architecture. Since every connection is single-level, and every component is MLS, a component cannot send data through a connection if the level of the data is different from that of the connection. Since communication is secure, any data received via a connection will have the same level as that of the connection. Given that every component properly enforces the two properties of the Bell-LaPadula model, so does the resulting architecture.

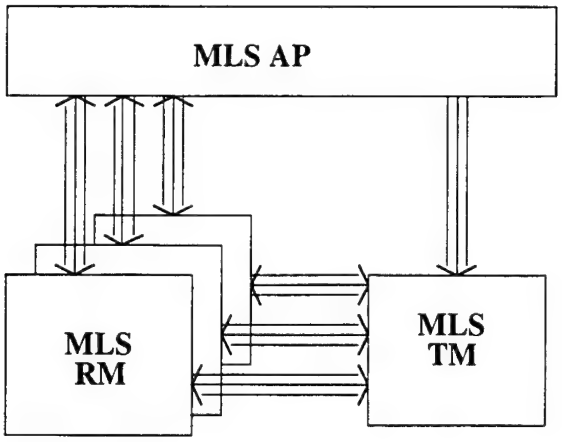

Figure 2.6: Secure DTP with Multilevel AP and RMs

2.6 Relating the SDTP Architectures

We can relate the four SDTP architectures by defining a general model for SDTP and then proving that each of the four SDTP architectures ia a correct specialization of it.

2.6.1 Correctness Criterion

The traditional interpretation of relative correctness is not strong enough for our purposes because it only requires that the concrete architecture extend the abstract architecture. Hence, a low-level architecture can introduce properties that contradict the higher-level architecture, which can lead to a violation of security properties in the reference architecture. For example, consider the proof in Section 2.5 that the SDTP architecture in Figure 2.5 is secure. The proof critically depends on the fact that the only way that AP and RMs communicate is through either the MLS TM or the MLS filter. The low-level architecture can extend this by including a new non-MLS flow from AP to an RM. This extended low-level architecture by definition "implements" the high-level architecture, but the extension clearly violates multilevel security requirements.

We have developed a new correctness criterion for architectures in which a concrete architecture implements *exactly* the abstract architecture, no more and no less [26]. In other words, a security hole that does not exist, with respect to a given security policy, in the abstract architecture will not come into existence in any concrete architecture. To use our earlier example, under our new correctness criteria, it is possible to ensure that no new data flow can be introduced in the low-level architecture. That is, if flow from B to C exists in the low-level architecture, then this flow must already exist in the high-level architecture, either as a direct or indirect flow. This flow thus implies that the

high-level design is not secure and should be modified. Conversely, if we can prove that the high-level architecture is secure, then we can rest assured that the mapped low-level architecture is also secure.

To formalize this notion, we first need to make explicit an important *completeness assumption* about architectures. In particular, we assume that an architecture contains *all* components, interfaces, and connections intended to be true of the architecture *at its level of detail.* If a fact is not explicit in the architecture, or deducible from it, we assume that it is not intended to be true of the architecture. In general, an architecture (whether static or dynamic) can contain an unbounded number of facts.

Formally, we need to prove that two architectures, represented as theories, are correct with respect to an interpretation mapping between them and the completeness assumption. Let Θ and Θ' be theories associated with an abstract and a concrete architecture, respectively. Let $\mathscr{I}$ be an interpretation mapping from the language of Θ to the language of Θ'. For every sentence φ, mapping $\mathscr{I}$ is a *theory interpretation* provided

$$\text{if } \varphi \in \Theta \text{ then } \mathscr{I}(\varphi) \in \Theta'$$

This is the usual definition of correctness.

Since a given architecture is assumed to be complete with respect to its level of detail, we additionally require that the concrete architecture add no new facts about the abstract architecture. To prove this, we must additionally show that

$$\text{if } \varphi \notin \Theta \text{ then } \mathscr{I}(\varphi) \notin \Theta'$$

in which case, $\mathscr{I}$ is a *faithful interpretation.* This says that, if a sentence is not in the abstract theory, its image cannot be in the concrete theory. (Observe that Θ' is a conservative extension of Θ provided the identity map faithfully interprets Θ in Θ'.) Theory Θ' *can* contain sentences not in $\mathscr{I}[\Theta]$ — i.e., new facts about the architecture not included in Θ — but these facts must not have any consequences expressible in the language of Θ.

2.6.2 The General Model

Let the AP, TM, and RMs be represented as functional components with labeled input and output ports connected by secure channels. More specifically, we require that

- Data have associated *security levels*, which are collections of authentication data and credentials that characterize the data's security standing. Thus, a datum can be thought of as a value together with a security label that determines its level.
- The ports where the data is supplied and received have associated *clearance levels.*
- The channels that carry data also have associated *clearance levels.*

The connections in the general model that represent the AR, XA, and TX interfaces are subject to three constraints.

1. An input port can receive data only if its clearance level dominates the data's security level.
2. An output port can supply data only if its clearance level is dominated by the data's security level.
3. A channel can carry data only if its clearance level dominates the data's security level.

If these constraints are satisfied, we may refer to ports as *secure ports* and to channels as *secure channels.*

2.6.3 Example Proof Relating Two Architectures

It can be proven that the four SDTP architectures correctly implement the general model. In this section, we focus on the most complicated proof, which involves Figure 2.4. Specifically, we prove that a combination of writing and reading data that is mediated by the MLS filter correctly implements secure dataflow.

This requires showing that a mapping $\mathscr{I}$ from the language of secure dataflow to the language of mediated reading and writing is a faithful interpretation of the theory of secure dataflow, Θ, in the theory of mediated reading and writing, Θ'. The interesting predicates in the language of secure dataflow are

- Secure_Channel(x), which means that x is a secure channel,
- Connects(x, y, z), which means that secure channel x connects secure output port y to z,
- Can_Carry(x, y), which means that secure channel x can carry datum y (i.e., that the datum contains an appropriate type value and has a security level no higher than the clearance level of the channel),
- Clearance_Level(x, y), which means that either the security label x is greater than or equal to the clearance level of y, when y is either an output port or channel, or the security label x is less than or equal to the clearance level of y, when y is either an input port, and
- Security_Level(x, y), which means that either the security label x is less than or equal to the security level of the datum y,

together with a lattice ordering $\leq$ of security labels.

The predicates of interest in the language of MLS-mediated reading and writing are

- MLS_Filter(x), which means that x is an MLS filter,

- Secure_Write(x, y), which means that x is a call site that performs a secure write to y,
- Secure_Read(x, y), which means that x is a call site that performs a secure read of y,
- Filter_Passes(x, y), which means that MLS filter x can pass datum y (i.e., that the datum contains an appropriate type value and has a security level no higher than the clearance level of the secure read and no lower than the clearance level of the secure write),
- Clearance_Level(x, y), which has the same meaning as in the secure dataflow language, and
- Security_Level(x, y), which also has the same meaning as in the secure dataflow language,

together with a lattice ordering of security labels which is also called $\leq$.

The relevant part of $\mathscr{I}$ can be defined as follows:

$$\begin{aligned}
&\mathscr{I}(\mathsf{Secure_Channel}(\mathbf{x})) \\
&\quad = \mathsf{MLS_Filter}(\mathscr{I}(\mathbf{x})) \\
&\mathscr{I}(\mathsf{Connects}(\mathbf{x}, \mathbf{y}, \mathbf{z})) \\
&\quad = [\mathsf{Secure_Write}(\mathscr{I}(\mathbf{y}), \mathscr{I}(\mathbf{x})) \\
&\qquad \wedge \mathsf{Secure_Read}(\mathscr{I}(\mathbf{z}), \mathscr{I}(\mathbf{x}))] \\
&\mathscr{I}(\mathsf{Can_Carry}(\mathbf{x}, \mathbf{y})) \\
&\quad = \mathsf{Filter_Passes}(\mathscr{I}(\mathbf{x}), \mathscr{I}(\mathbf{y})) \\
&\mathscr{I}(\mathsf{Clearance_Level}(\mathbf{x}, \mathbf{y})) \\
&\quad = \mathsf{Clearance_Level}(\mathscr{I}(\mathbf{x}), \mathscr{I}(\mathbf{y})) \\
&\mathscr{I}(\mathsf{Security_Level}(\mathbf{x}, \mathbf{y})) \\
&\quad = \mathsf{Security_Level}(\mathscr{I}(\mathbf{x}), \mathscr{I}(\mathbf{y})) \\
&\mathscr{I}(\mathbf{x} \leq \mathbf{y}) = [\mathscr{I}(\mathbf{x}) \leq \mathscr{I}(\mathbf{y})]
\end{aligned}$$

together with clauses that map each abstract-level secure channel to a concrete-level mediated read and write combination.

Note that $\mathscr{I}$ naturally determines a mapping $\mathscr{I}^{\partial}$ from structures of the *concrete* language to structures of the *abstract* language. Given a structure $\mathfrak{M}'$ of the concrete language, $\mathscr{I}^{\partial}(\mathfrak{M}')$ is defined as follows. The universe of $\mathscr{I}^{\partial}(\mathfrak{M}')$ is the same as $|\mathfrak{M}'|$, the universe of $\mathfrak{M}'$. If $\mathscr{I}$ maps atomic formula $\mathbf{P}(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ to concrete formula φ, then the extension of $\mathbf{P}$ in $\mathscr{I}^{\partial}(\mathfrak{M}')$ is the set of n-tuples from the universe of $\mathfrak{M}'$ that satisfy φ. That is, the extension of $\mathbf{P}$ in $\mathscr{I}^{\partial}(\mathfrak{M}')$ is

$$\begin{aligned}
&\{\langle m_1, m_2, \ldots, m_n\rangle \in |\mathfrak{M}'|^n : \\
&\quad \mathfrak{M}' \vDash \varphi[\mathbf{x}_1/m_1, \mathbf{x}_2/m_2, \ldots, \mathbf{x}_n/m_n]\}
\end{aligned}$$

Showing that $\mathscr{I}$ is a theory interpretation is simply a matter of formally deriving the concrete level interpretation of each abstract level axiom from the concrete level axioms. For example, the image of the security constraint

$$\begin{aligned}
&\forall x\, \forall y\, \forall z\, \forall w \\
&\quad [\mathsf{Secure_Channel}(x) \\
&\qquad \wedge \mathsf{Clearance_Level}(x, y) \\
&\qquad \wedge \mathsf{Datum}(z) \\
&\qquad \wedge \mathsf{Security_Level}(z, w) \\
&\qquad \wedge \mathsf{Can_Carry}(x, z) \\
&\qquad\qquad \rightarrow z \leq y]
\end{aligned}$$

must be derived from the axioms describing the properties of the filter. This is straightforward.

To show that $\mathscr{I}$ is faithful, we will describe a mapping $\mathscr{J}$ from models of abstract-level theory Θ to models of concrete-level theory Θ' such that, for every model $\mathfrak{M}$ of Θ, $\mathscr{I}^{\partial}(\mathscr{J}(\mathfrak{M}))$ is isomorphic to $\mathfrak{M}$. The faithfulness of $\mathscr{I}$ follows, by the model-theoretic result cited in [26].

As the definition of $\mathscr{I}$ suggests, the only difficulty in "inverting" the interpretation is handling the equation for Connects. But is it easy to see that letting the extension of Secure_Read in $\mathscr{J}(\mathfrak{M})$ be

$$\begin{aligned}
&\{\langle m_1, m_2\rangle \in |\mathfrak{M}|^2 : \text{for some } m_3 \text{ in } |\mathfrak{M}|, \\
&\quad \mathfrak{M} \vDash \mathsf{Connects}(x_1, x_2, x_3)[x_1/m_2, x_2/m_3, x_3/m_1]\}
\end{aligned}$$

and letting the extension of Secure_Write in $J(\mathfrak{M})$ be

$$\begin{aligned}
&\{\langle m_1, m_2\rangle \in |\mathfrak{M}|^2 : \text{for some } m_3 \text{ in } |\mathfrak{M}|, \\
&\quad \mathfrak{M} \vDash \mathsf{Connects}(x_1, x_2, x_3)[x_1/m_2, x_2/m_1, x_3/m_3]\}
\end{aligned}$$

will yield the required structure.

2.7 Chapter Summary

We have combined results from the software and security research communities to form a new methodology for the construction of secure architectures. The method involves the formalization of a system architecture with security mechanisms embedded directly in the architecture. More specifically, the mechanisms are intended to provide secure access control as defined by the Bell-LaPadula model (the simple security property and the $\star$–property). A proof about the security of a system implementation is performed by reasoning about its architecture. If an architecture is secure, every valid instance of it is secure. Architecture instantiation is equivalent semantically to theory instantiation.

An important contribution of our work is the modeling of a system in terms of multiple secure architectures that are related by formal mappings. The architectures may represent both horizontal and vertical decompositions. Proper

application of our modeling technique enables vendors to develop single- or multilevel products with little or no knowledge about the overall application. Similarly, customers can select single- or multilevel components, knowing with confidence that the security of the overall system will be intact. An example of this was seen in the SDTP development, where the classification of a component had a radical effect on the SDTP architecture. Our modeling technique also offers benefits to the system architect, who can initially develop a simple, abstract system architecture that is easy to reason about but is too restrictive for implementation purposes. We saw this in the abstract SDTP architecture, which requires that all components provide multilevel access control. The gap between it and the three more concrete system architectures, which are less restrictive with respect to security, was bridged by refinement mappings that were shown to preserve the security property of the abstract architecture.

It is important to mention that our approach to architecture modeling is application independent. In this paper, we concentrated on the X/Open DTP reference architecture to maximize the commercial relevance of our work. However, our approach is not tied to any particular application and should apply equally well in the development of other secure architectures.

Future work includes the development of standard refinement rules for implementing secure architectures in execution environments containing existing or emerging security standards. It also includes an extension of SADL to model behavior-related security properties using the RAPIDE architecture prototyping language. This would make it possible, for example, to reason about covert channels in terms of a system architecture.

Chapter 3

Proving Faithfulness

In an earlier paper on the foundations of correct architecture refinement [26], my colleagues and I gave a model-theoretic characterization of faithfulness. Although I cannot claim that this characterization is original, I have been unable to find it stated, much less justified, in standard references on model theory [6, 8, 15, 24, 44].[1] This paper will provide the justification.[2]

I will assume that the reader is familiar with basic logical concepts, such as *satisfaction*, *model*, *validity*, and *consistency*. Note in particular that a *sentence* is a formula containing no free occurrences of variables, and that a *theory* is a set of sentences that is closed under consequence. The logical notation used below is generally standard or self-explanatory. For example,

$$\mathfrak{A} \models \varphi\,[\mathbf{v}_0/a_0, \mathbf{v}_1/a_1, \ldots, \mathbf{v}_{n-1}/a_{n-1}],$$

which will usually be shortened to

$$\mathfrak{A} \models \varphi\,[a_0, a_1, \ldots, a_{n-1}]$$

when the identities of the free variables of φ are unimportant (or made clear by the context), means that the assignment of values $a_0, a_1, \ldots, a_{n-1}$ in the domain of the structure $\mathfrak{A}$ to the free variables $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ of the formula φ satisfies φ in $\mathfrak{A}$, while

$$\mathfrak{A} \models \varphi(\mathbf{v}_0/\mathbf{t}_0, \mathbf{v}_1/\mathbf{t}_1, \ldots, \mathbf{v}_{n-1}/\mathbf{t}_{n-1}),$$

or

$$\mathfrak{A} \models \varphi(\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{n-1})$$

for short, means the formula that results when terms $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{n-1}$ are simultaneously substituted for free occurrences of the variables $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ in

[1] One reference [44, p. 96, Problem 9c] does give a special case as an exercise.

[2] We originally intended to include this argument in an appendix to the earlier paper, but decided that most readers of the journal in which the paper appeared would not find it helpful.

the formula φ (renaming bound variables as necessary to avoid capture of the free variables of $\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_{n-1}$) is true in the structure $\mathfrak{A}$. The prefix "$\mathscr{L}$-" will frequently be used to emphasize the elementary language $\mathscr{L}$ under consideration.[3] For example, saying, that φ is an $\mathscr{L}$-formula means that all the parameters of φ are among the parameters of $\mathscr{L}$. I also assume that the reader is familar with the fundamentals of logical metatheory, including the Completeness and Compactness Theorems for first-order logic and the Craig Interpolation Theorem. All this logical background can be found in any good logic textbook, such as the standard model theory references cited above.

For our purposes,[4] it is convenient to consider an interpretation of elementary language $\mathscr{L}$ in elementary theory Θ' to be a mapping $\mathscr{I}$ from $\mathscr{L}$-formulas to $\mathscr{L}'$-formulas that (1) "respects logic", in the sense that, for all $\mathscr{L}$-formulas φ and ψ, and every variable $\mathbf{v}$,

$$\begin{aligned}
\mathscr{I}(\neg\varphi) &= \neg\mathscr{I}(\varphi), \\
\mathscr{I}(\varphi \wedge \psi) &= \mathscr{I}(\varphi) \wedge \mathscr{I}(\psi), \\
\mathscr{I}(\varphi \vee \psi) &= \mathscr{I}(\varphi) \vee \mathscr{I}(\psi), \\
\mathscr{I}(\varphi \rightarrow \psi) &= \mathscr{I}(\varphi) \rightarrow \mathscr{I}(\psi), \\
\mathscr{I}(\forall \mathbf{v}\, \varphi) &= \forall \mathbf{v}\, [\omega_{\mathscr{I}}(\mathbf{v}) \rightarrow \mathscr{I}(\varphi)], \text{and} \\
\mathscr{I}(\exists \mathbf{v}\, \varphi) &= \exists \mathbf{v}\, [\omega_{\mathscr{I}}(\mathbf{v}) \wedge \mathscr{I}(\varphi)].
\end{aligned}$$

where $\omega_{\mathscr{I}}(\mathbf{v})$ is an $\mathscr{L}'$-formula whose only free variable is $\mathbf{v}$, and (2) satisfies the following conditions:

$$\Theta' \vDash \exists \mathbf{v}\, \omega_{\mathscr{I}}(\mathbf{v}),$$

and, for every individual parameter $\mathbf{a}$ of $\mathscr{L}$,

$$\Theta' \vDash \exists \mathsf{x}_1\, \forall \mathsf{x}_0\, [\, \mathscr{I}(\mathsf{x}_0 = \mathbf{a}) \leftrightarrow \mathsf{x}_0 = \mathsf{x}_1\,].$$

An interpretation $\mathscr{I}$ of $\mathscr{L}$ in $\mathscr{L}'$-theory Θ' is an *interpretation of* $\mathscr{L}$-theory Θ *in* Θ' iff, for every $\mathscr{L}$-sentence φ,

$$\Theta \vDash \varphi \implies \Theta' \vDash \mathscr{I}(\varphi).$$

An interpretation $\mathscr{I}$ of Θ in Θ' is *faithful* iff, for every $\mathscr{L}$-sentence φ,

$$\Theta' \vDash \mathscr{I}(\varphi) \implies \Theta \vDash \varphi.$$

As a first step toward proving the main result, consider the notion of an *elementary embedding* of one structure in another. For any relational language $\mathscr{L}$, an elementary embedding of $\mathscr{L}$-structure $\mathfrak{A}$ in $\mathscr{L}$-structure $\mathfrak{B}$ is a function f from the domain of $\mathfrak{A}$ to the domain of $\mathfrak{B}$ such that, for any $\mathscr{L}$-formula φ

[3] Since only pure (i.e., non-logical constant-free), function parameter-free, elementary languages will be considered below, languages can be identified with the set of their predicate and individual parameters.

[4] Different authors define *interpretation* somewhat differently. Any of the standard textbook definitions would be adequate for our purposes.

and any assignment of values $a_0, a_1, \ldots, a_{n-1}$ in the domain of $\mathfrak{A}$ to the free variables of φ,

$$\mathfrak{A} \models \varphi[a_0, a_1, \ldots, a_{n-1}] \quad \text{iff} \quad \mathfrak{B} \models \varphi[f(a_0), f(a_1), \ldots, f(a_{n-1})]$$

If $\mathfrak{A}$ can be elementarily embedded in $\mathfrak{B}$, then $\mathfrak{A}$ and $\mathfrak{B}$ "agree" on the extension of all formulas. In that case, they certainly agree on the truth value of all sentences, i.e., they are elementarily equivalent.

The *description*, or *elementary diagram*, $\mathrm{Ds}(\mathfrak{A})$, of an $\mathscr{L}$-structure $\mathfrak{A}$ with domain A is obtained as follows: expand $\mathscr{L}$ so that it contains a name $\underline{a}$ for every member a of A;[5] expand $\mathfrak{A}$ to a structure $(\mathfrak{A}, a)_{a \in A}$ for the language $\mathscr{L} \cup \underline{A}$, where $\underline{A} = \{\underline{a} : a \in A\}$, by assigning each name $\underline{a}$ to the corresponding element a; let $\mathrm{Ds}(\mathfrak{A})$ be the set of $\mathscr{L} \cup \underline{A}$-sentences true in $(\mathfrak{A}, a)_{a \in A}$. For any $\mathscr{L}$-structure $\mathfrak{A}$, any $\mathscr{L}$-formula φ, and any assignment of values $a_0, a_1, \ldots, a_{n-1}$ in A to the free variables of φ,

$$\mathfrak{A} \models \varphi[a_0, a_1, \ldots, a_{n-1}] \quad \text{iff} \quad (\mathfrak{A}, a)_{a \in A} \models \varphi(\underline{a}_0, \underline{a}_1, \ldots, \underline{a}_{n-1})$$

So the connection between elementary embeddings and descriptions is straightforward: $\mathfrak{A}$ can be elementarily embedded in $\mathfrak{B}$ iff $\mathfrak{B}$ can be expanded to an $\mathscr{L} \cup \underline{A}$-structure that models $\mathrm{Ds}(\mathfrak{A})$.

The bulk of the proof will consist in establishing two lemmas. The first lemma establishes a relationship between *descriptions of structures* and *faithful interpretations.*

Lemma 1: For any faithful interpretation $\mathscr{I}$ of consistent $\mathscr{L}$-theory Θ in $\mathscr{L}'$-theory Θ' and any model $\mathfrak{A}$ of Θ with domain A, the set of sentences $\mathscr{I}[\mathrm{Ds}(\mathfrak{A})] \cup \Theta'$ is consistent, where $\mathscr{I}$ is extended from $\mathscr{L}$ to $\mathscr{L} \cup \underline{A}$ by "mapping each new name $\underline{a}$ to itself",[6] i.e., for every atomic $\mathscr{L}$-formula φ and every substitution of names $\underline{a}_0, \underline{a}_1, \ldots, \underline{a}_{n-1}$ in $\underline{A}$ for the free variables $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}$ of φ, let $\mathscr{I}$ map atomic $\mathscr{L} \cup \underline{A}$-formula

$$\varphi(\mathbf{v}_0/\underline{a}_0, \mathbf{v}_1/\underline{a}_1, \ldots, \mathbf{v}_{n-1}/\underline{a}_{n-1})$$

[5] Choose fresh names, so that no $\underline{a}$ is in any other language under consideration.

[6] Note that $\varphi(\mathbf{v}_0/\underline{a}_0, \mathbf{v}_1/\underline{a}_1, \ldots, \mathbf{v}_{n-1}/\underline{a}_{n-1})$ is logically equivalent to

$$\exists \mathbf{v}_0 \exists \mathbf{v}_1 \ldots \exists \mathbf{v} \left[\bigwedge_{i<n} \mathbf{v}_i = \underline{a}_i \wedge \varphi \right], \tag{3.1}$$

that formula (3.1) is mapped to

$$\exists \mathbf{v}_0 \exists \mathbf{v}_1 \ldots \exists \mathbf{v} \left[\bigwedge_{i<n} \omega_{\mathscr{I}}(\mathbf{v}_i) \wedge \bigwedge_{i<n} \mathbf{v}_i = \underline{a}_i \wedge \mathscr{I}(\varphi) \right] \tag{3.2}$$

by $\mathscr{I}$ if every equation $\mathbf{v}_i = \underline{a}_i$ is mapped to itself, and that formula (3.2) is logically equivalent to

$$\bigwedge_{i<n} \omega_{\mathscr{I}}(\underline{a}_i) \wedge (\mathscr{I}(\varphi))(\mathbf{v}_0/\underline{a}_0, \mathbf{v}_1/\underline{a}_1, \ldots, \mathbf{v}_{n-1}/\underline{a}_{n-1}).$$

to $\mathscr{L}' \cup \underset{\sim}{A}$-formula

$$\bigwedge_{i<n} \omega_{\mathscr{I}}(\underset{\sim}{a}_i) \wedge \big(\mathscr{I}(\varphi)\big)(\mathbf{v}_0/\underset{\sim}{a}_0, \mathbf{v}_1/\underset{\sim}{a}_1, \ldots, \mathbf{v}_{n-1}/\underset{\sim}{a}_{n-1}).$$

I will first present a proof of this lemma for the special case of conservative extensions — recall that $\mathscr{L}'$-theory Θ' is a conservative extension of $\mathscr{L}$-theory Θ iff the identity mapping on $\mathscr{L}$-sentences is a faithful interpretation of Θ in Θ' — in order to highlight the essential ideas. Then I will explain how the proof is modified to handle other interpretations.

Suppose, for the purpose of deriving a contradiction, that $\mathrm{Ds}(\mathfrak{A}) \cup \Theta'$ is inconsistent. Intuitively, there must be an $\mathscr{L}$-sentence φ such that $\mathrm{Ds}(\mathfrak{A})$ implies that φ is false and Θ' implies that φ is true. After all, $\mathrm{Ds}(\mathfrak{A})$ says nothing in particular about the additional symbols of $\mathscr{L}'$ that are not in $\mathscr{L}$, and Θ' says nothing in particular about the names $\underset{\sim}{a}$ that were added to $\mathscr{L}$ to obtain $\mathscr{L} \cup \underset{\sim}{A}$; the disagreement between the two theories that is responsible for the inconsistency should be expressible in the common part of their languages. This intuition can be shown to be correct by employing the metatheory of first-order logic. According to the Compactness Theorem, there is a finite subset Δ of $\mathrm{Ds}(\mathfrak{A})$ and a finite subset Γ of Θ' such that $\Delta \cup \Gamma$ is inconsistent. Equivalently,

$$\vDash \bigwedge \Gamma \rightarrow \neg \bigwedge \Delta$$

By the Completeness Theorem,

$$\vdash \bigwedge \Gamma \rightarrow \neg \bigwedge \Delta$$

Therefore, by the Craig Interpolation Theorem, there is an $\mathscr{L}$-sentence φ such that

$$\vdash \bigwedge \Gamma \rightarrow \varphi \quad \text{and} \quad \vdash \varphi \rightarrow \neg \bigwedge \Delta$$

Equivalently, by the Soundness Theorem,

$$\Gamma \vDash \varphi \quad \text{and} \quad \Delta \vDash \neg\varphi$$

and so

$$\varphi \in \Theta' \quad \text{and} \quad \neg\varphi \in \mathrm{Ds}(\mathfrak{A})$$

Since $\mathscr{L}$-sentence φ is in Θ' and the extension is conservative, φ is in Θ. Since $\mathfrak{A}$ is a model of Θ, φ is true in $\mathfrak{A}$ and so is among the sentences in $\mathrm{Ds}(\mathfrak{A})$. But this is impossible, as φ was chosen so that $\neg\varphi$ is in $\mathrm{Ds}(\mathfrak{A})$ and the description of a structure is consistent by definition. Because the supposition that $\mathrm{Ds}(\mathfrak{A}) \cup \Theta'$ is inconsistent led to a contradiction, it follows that $\mathrm{Ds}(\mathfrak{A}) \cup \Theta'$ is consistent.

The proof for interpretations other than the identity is much the same. If we had used the more restrictive notion of *interpretation* where predicate parameters are mapped to predicate parameters and individual parameters to individual parameters [44], essentially the same argument used above would work.

But this argument breaks down when our more liberal definition is employed, because interpretations of theories need not be theories, and so a consequence of $\mathscr{I}[\mathrm{Ds}(\mathfrak{A})]$ in which no $\underline{a}$ occurs may not be the image of an $\mathscr{L}$-formula under $\mathscr{I}$.[7] However, it is easy to see that if we replace the target language $\mathscr{L}'$ with $\mathscr{L} \uplus \mathscr{L}'$ and also replace $\mathscr{I}[\mathrm{Ds}(\mathfrak{A})]$ by the result of adding both, for every n-ary predicate $\mathbf{P}$ of $\mathscr{L}$, the axiom

$$\forall \mathbf{v}_0 \forall \mathbf{v}_1 \ldots \forall \mathbf{v}_{n-1} [\, \mathbf{P}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}) \leftrightarrow \mathscr{I}(\mathbf{P}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}))\,]$$

and, for every individual parameter $\mathbf{c}$ of $\mathscr{L}$, the axiom

$$\forall \mathbf{v}_0 [\, \mathbf{v}_0 \equiv \mathbf{c} \leftrightarrow \mathscr{I}(\mathbf{v}_0 \equiv \mathbf{c})\,]$$

to a version of $\mathrm{Ds}(\mathfrak{A})$ in which quantifiers are bounded by $\omega_{\mathscr{I}}$, the consistency of the union with Θ' is unaffected. Clearly, the two "say the same thing about the world", in somewhat different languages. So, at least for the purpose of this argument, an interpretation of Θ in Θ' can be viewed as an interpretation in the more restrictive sense of Θ in a definitional extension of Θ'.

The second lemma says that interpretation mappings "preserve meaning".

Lemma 2: For every interpretation $\mathscr{I}$ of $\mathscr{L}$-theory Θ in $\mathscr{L}'$-theory Θ', every model $\mathfrak{A}'$ of Θ', every $\mathscr{L}$ formula φ, and every assignment of values $a_0, a_1, \ldots, a_{n-1}$ in the domain of $\mathfrak{A}'$ to the variables of φ,

$$\mathscr{I}^{\partial}(\mathfrak{A}') \vDash \varphi\,[a_0, a_1, \ldots, a_{n-1}] \quad \text{iff} \quad \mathfrak{A}' \vDash \mathscr{I}(\varphi)\,[a_0, a_1, \ldots, a_{n-1}]$$

where $\mathscr{I}^{\partial}$ is the natural mapping from $\mathscr{L}'$-structures to $\mathscr{L}$-structures determined by $\mathscr{I}$: the domain of $\mathscr{I}^{\partial}(\mathfrak{A}')$ is the same as the domain of $\mathfrak{A}'$, the denotation of n-ary predicate $\mathbf{P}$ of $\mathscr{L}$ in $\mathscr{I}^{\partial}(\mathfrak{A}')$ is

$$\{\langle a_0, a_1, \ldots, a_{n-1}\rangle : \mathfrak{A}' \vDash \mathscr{I}(\mathbf{P}(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{n-1}))\,[a_0, a_1, \ldots, a_{n-1}]\}$$

and the denotation of the individual parameter $\mathbf{c}$ of $\mathscr{L}$ in $\mathscr{I}^{\partial}(\mathfrak{A}')$ is the a such that $\mathfrak{A}' \vDash \mathscr{I}(\mathbf{v} \equiv \mathbf{c})\,[a]$.

Thanks to the absence of function symbols, this lemma can be proved by a straightforward induction on formulas.[8] For atomic formulas, the result is an immediate consequence of the definition of $\mathscr{I}^{\partial}$. If φ is, say, a conjunction $\psi \wedge \chi$,

[7] For example, if Θ is the set of consequences of the $\{\mathbf{P}\}$-formula $\forall \mathbf{v}\, \mathbf{P}(\mathbf{v})$ and $\mathscr{I}$ maps $\mathbf{P}(\mathbf{v})$ to $\mathbf{Q}(\mathbf{v}) \wedge \mathbf{R}(\mathbf{v})$, then $\forall \mathbf{v}\, [\, \omega_{\mathscr{I}}(\mathbf{v}) \rightarrow \mathbf{Q}(\mathbf{v})\,]$ is a consequence of $\mathscr{I}[\Theta]$ but is not the image of any $\{\mathbf{P}\}$-sentence under $\mathscr{I}$.

[8] Inclusion of function symbols in the language would merely require that the base case of our induction, the atomic formulas, be established by induction on the number of function symbols that occur in atoms.

then

$\mathscr{I}^{\partial}(\mathfrak{A}') \models (\psi \wedge \chi)\,[a_0, a_1, \ldots, a_{n-1}]$

iff $\mathscr{I}^{\partial}(\mathfrak{A}') \models \psi\,[a_0, a_1, \ldots, a_{n-1}]$ and $\mathscr{I}^{\partial}(\mathfrak{A}') \models \chi\,[a_0, a_1, \ldots, a_{n-1}]$
(By the definition of *satisfaction*)

iff $\mathfrak{A}' \models \mathscr{I}(\psi)\,[a_0, a_1, \ldots, a_{n-1}]$ and $\mathfrak{A}' \models \mathscr{I}(\chi)\,[a_0, a_1, \ldots, a_{n-1}]$
(By the induction hypothesis)

iff $\mathfrak{A}' \models (\mathscr{I}(\psi) \wedge \mathscr{I}(\chi))\,[a_0, a_1, \ldots, a_{n-1}]$
(By the definition of *satisfaction*)

iff $\mathfrak{A}' \models \mathscr{I}(\psi \wedge \chi)\,[a_0, a_1, \ldots, a_{n-1}]$
(By the definition of *interpretation*)

The argument for the other connectives and quantifiers is similar.

The main result is an easy consequence of the lemmas.

Theorem: For any interpretation $\mathscr{I}$ of elementary $\mathscr{L}$-theory Θ in elementary $\mathscr{L}'$-theory Θ', the following are equivalent:

(1) $\mathscr{I}$ is faithful;

(2) for every model $\mathfrak{A}$ of Θ there is a model $\mathfrak{A}'$ of Θ' such that $\mathscr{I}^{\partial}(\mathfrak{A}')$ can be expanded to a model of $\mathrm{Ds}(\mathfrak{A})$;

(3) for every model $\mathfrak{A}$ of Θ there is a model $\mathfrak{A}'$ of Θ' such that $\mathfrak{A}$ can be elementarily embedded in $\mathscr{I}^{\partial}(\mathfrak{A}')$.

Proof that (1) implies (2): Let $\mathfrak{A}$ be a model of Θ. According to Lemma 1, $\mathscr{I}[\mathrm{Ds}(\mathfrak{A})] \cup \Theta'$ is consistent. Let $\mathfrak{A}'$ be the reduct of some model $\mathfrak{B}$ of this theory to $\mathscr{L}'$. The structure $\mathscr{I}^{\partial}(\mathfrak{B})$, which is an expansion of $\mathscr{I}^{\partial}(\mathfrak{A}')$ to $\mathscr{L} \cup \underline{A}$, is a model of $\mathrm{Ds}(\mathfrak{A})$, by Lemma 2.

Proof that (2) implies (3): This follows from the connection between elementary embeddings and descriptions noted above.

Proof that (3) implies (1): Let φ be an $\mathscr{L}$-sentence such that $\mathscr{I}(\varphi)$ is in Θ'. Suppose that $\mathfrak{A}$ is a model of Θ. By hypothesis, there is a model $\mathfrak{A}'$ of Θ' such that $\mathfrak{A}$ can be elementarily embedded in $\mathscr{I}^{\partial}(\mathfrak{A}')$. Since $\mathfrak{A}'$ is a model of Θ' and $\mathscr{I}(\varphi)$ is in Θ', $\mathscr{I}(\varphi)$ is true in $\mathfrak{A}'$. But then, using Lemma 2, φ is true in $\mathscr{I}^{\partial}(\mathfrak{A}')$, which is elementarily equivalent to $\mathfrak{A}$. Generalizing, φ is true in every model of Θ, and hence is a member of Θ.

Intuitively, the theorem can be thought of as saying that an interpretation $\mathscr{I}$ of first-order theory Θ in first-order theory Θ' is faithful iff, for every model $\mathfrak{A}$ of Θ, there is a model $\mathfrak{A}'$ of Θ' such that, when the "abstraction map" $\mathscr{I}^{\partial}$ is applied to $\mathfrak{A}'$, the structure that results cannot be distinguished from $\mathfrak{A}$ using the resources of first-order logic. This suggests that analogous results can be established when a different logic is used to formalize the architectures. For example, the logic L^{ω}, often called *higher-order logic* or *simple type theory*, can

characterize any structure up to isomorphism.[9] One would therefore expect that an interpretation $\mathscr{I}$ of an L^{ω} theory Θ in an L^{ω} theory Θ' is faithful iff, for every model $\mathfrak{A}$ of Θ, there is a model $\mathfrak{A}'$ of Θ' such that $\mathscr{I}^{\delta}(\mathfrak{A}')$ is isomorphic to $\mathfrak{A}$. It is easy to prove that this expectation is correct. Formulating and proving analogues of the theorem for other logics of interest is left as an exercise.

[9] L^{ω} allows quantification over relations among individuals (or, equivalently, functions from individuals to individuals), relations among relations among individuals (or functions from functions on individuals to functions on individuals), and so on.

Chapter 4

Correctness of Architectural Refinements: a simplified approach

4.1 Two Approachs to Establishing Correctness

In a previous paper [26], my colleagues and I presented an approach to proving correctness of architectural refinement patterns. A correspondence between architectural specifications, such as the high-level compiler specification in Figure 4.1, and theories in first-order logic was defined in terms of a mapping from specification elements to axioms for the theories.[1] For example, the presence of the dataflow connector that carries objects of type AST (i.e., abstract syntax trees) from the parser component to the analyzer/optimizer component in Figure 4.1 corresponds to the axioms

$$\mathsf{Channel}(\mathsf{ast\_intermediate})$$
$$\forall x_0\,[\,\mathsf{AST}(x_0) \rightarrow \mathsf{Can\_Carry}(\mathsf{ast\_intermediate}, x_0)\,]$$

The theory that corresponds to a specification is simply the set of all consequences of the axioms obtained from the elements of the specification, together with general axioms that constrain the meanings of the component, port, and connector predicates that appear in the specification.

This theory is rather weak, in the sense that it does not determine the truth value of many sentences in the language. For example, it does not contain any explicit "extremal axioms" to preclude the existence of additional objects not

[1] Strictly speaking, the content of the informal dataflow diagram was formalized in a textual specification language. For the purposes of this paper, let us eliminate the middleman and pretend that diagrammatic representations are sufficiently precise to serve as formal architectural specifications.

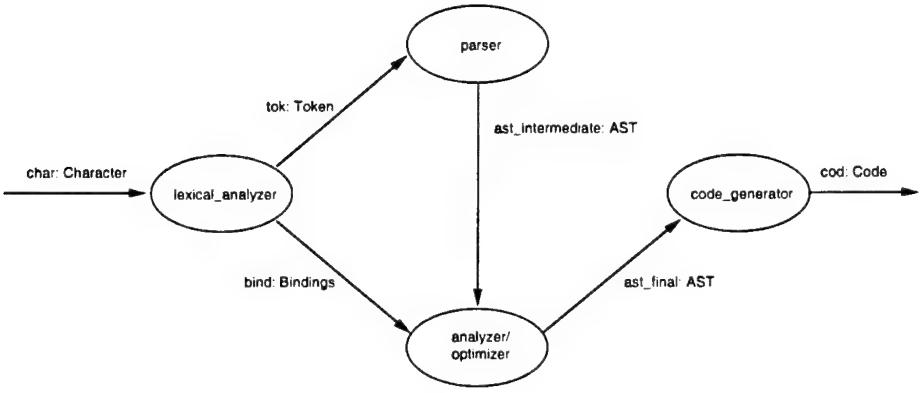

Figure 4.1: Dataflow Architecture for a Compiler

mentioned in the specification. Neither the sentence

$$\neg\,\exists x_0 \exists x_1 \exists x_2\,[\,\mathsf{Out\_Port}(x_1, \mathsf{parser}) \\ \wedge\, \mathsf{In\_Port}(x_2, \mathsf{lexical\_analyzer}) \wedge \mathsf{Connects}(x_0, x_1, x_2)\,]$$

which says that there is no dataflow channel from the parser to the lexical analyzer, nor its negation, is a consequence of the theory that corresponds to the architecture in Figure 4.1. Of course, that no such dataflow channel is shown means that the specifier intended that no such channel be implemented (or, more accurately, that no such channel should be observable at this level of abstraction). Some of these "truth value gaps" can be eliminated by adding axioms, but perhaps not all: the theory of an inductive datatype, such as AST, may well be essentially incomplete. Nonetheless, our theories are treated as if they were complete, in the sense that our criterion for correct refinement is that the higher-level theory must be *faithfully* interpretable in the lower-level theory.[2] In other words, the lower-level theory must not add any detail that could have been expressed at the higher level, such as the existence of additional unmentioned objects in the higher-level ontology.

But there is an alternative, perhaps more natural, logical interpretation of the dataflow diagram in Figure 4.1. Any application of mathematics requires construction of a mathematical model of some phenomena of interest. This mathematical model must share relevant structure with the phenomena, so that

[2]Recall that an interpretation $\mathscr{I}$ of the language of theory Θ in the theory Θ' is an interpretation of Θ in Θ' iff for every sentence φ in the language of Θ,

$$\varphi \in \Theta \implies \mathscr{I}(\varphi) \in \Theta'$$

If, in addition, for every sentence φ in the language of Θ,

$$\mathscr{I}(\varphi) \in \Theta' \implies \varphi \in \Theta$$

then $\mathscr{I}$ is said to be *faithful*.

conclusions derived from reasoning about the structure of the model are true of the structure of the phenomena being modeled as well. Examples are ubiquitous in computing. For example, one might attempt to derive truths about the behavior of a particular parsing program running on a particular machine by reasoning about a finite state automaton that mathematically models it. It seems very natural to suppose that this is exactly the function a system specification, whether behavioral or architectural, is supposed to perform. In short, from this alternative perspective, *architectural specifications are intended to be mathematical models of the systems they specify.*

Just as a finite state automaton, which is formally defined as some sort of mathematical structure, can be depicted as a collection of boxes and arrows, a system architecture diagram can be construed as a depiction of a particular mathematical structure. Consider once again the dataflow diagram of Figure 4.1. The similarity type of the structure that this diagram depicts is determined by its architectural style. Since this is a dataflow diagram, the structure must provide the extensions for the various predicate parameters of the dataflow language — Channel, AST, Can_Carry, and so on — and must also identify the particular individuals denoted by the individual parameters of the specification, such as parser. (Note that this structure is not finite. Most often, the number of components and connectors in specifications will be finite,[3] but the datatypes are often infinite.)

In the "model-based" treatment of specifications, the logical theory that corresponds to the specification is simply the theory of the structure that the specification depicts. This theory is, of course, complete, which is important because *every standard interpretation*[4] *of a complete theory in a consistent theory is faithful.* The proof of this fact is easy: for any standard interpretation $\mathscr{I}$ of complete theory Θ in consistent theory Θ' and any sentence φ of the language of Θ,

$$\begin{aligned}
\varphi \notin \Theta &\implies \neg\varphi \in \Theta && (\Theta \text{ is complete}) \\
&\implies \mathscr{I}(\neg\varphi) \in \Theta' && (\mathscr{I} \text{ interprets } \Theta \text{ in } \Theta') \\
&\implies \neg\mathscr{I}(\varphi) \in \Theta' && (\text{St'd interp'ns preserve logic}) \\
&\implies \mathscr{I}(\varphi) \notin \Theta' && (\Theta' \text{ is consistent})
\end{aligned}$$

and so, universally generalizing the contrapositive, $\mathscr{I}$ is faithful.

In the "property-oriented" treatment of our previous paper, proving that the standard interpretation associated with a candidate refinement is a theory interpretation was easy. A standard interpretation maps derivations to derivations.[5] Therefore, if finite axiomatizations of Θ and Θ' are available, then it can be shown that $\mathscr{I}$ interprets Θ in Θ' by deriving the image of each axiom of

[3] However, it might sometimes be convenient to model a conceptually unbounded number of components or connectors by an infinite number of components or connectors, just as a Turing machine's infinite tape is used to model conceptually unbounded memory.

[4] See the Section 4.2 for the definition of *standard interpretation.*

[5] Actually, the derivations that result can contain small "gaps" due to the introduction of bounds on quantifiers, but filling these in is trivial.

Θ under $\mathscr{I}$ from the axioms of Θ'. But proving faithfulness is harder. A substantial model-theoretic result served as the basis of the method we proposed. So the fact that faithfulness is automatic on the new approach makes it look quite promising. Unfortunately, there is no simple, mechanical way to obtain axioms for the theory of a structure from a representation of that structure. Indeed, the theory of the structure corresponding to some specifications may very well have no recursive, much less finite, axiomatization. Some other way is required for recognizing when a basis mapping from the parameters of the language of one structure to the language of another structure determines a standard interpretation of the theory of the former struture in the theory of the latter structure.

4.2 Brief Introduction to Interpretations

Two different sorts of definition of the term *interpretation* appear in the literature. Logic textbooks [8, 15, 44] provide rather concrete definitions, explaining in some detail how mappings from the symbols of one theory to the expressions of another can be extended to mappings from sentences to sentences and exploring the properties of those mappings. For more advanced purposes [2, pp. 470–471], more abstract definitions are often preferred. The main advantage of a more abstract approach is flexibility. Many more mappings can count as interpretations if an abstract approach is adopted. But flexibility has a corresponding cost. Most important, if an interpretation is not defined inductively on formulas, then inductive proofs of properties of the interpretation become much more complicated.

For our applications, considerable flexibility is occasionally required [41]. Therefore, our definition is maximally abstract: an *interpretation* $\mathscr{I}$ of the theory Θ in the theory Θ' is a (total) mapping from the sentences of the language $\mathscr{L}$ of Θ to the sentences of the language $\mathscr{L}'$ of Θ' such that, for every $\mathscr{L}$-sentence φ,

$$\varphi \in \Theta \implies \mathscr{I}(\varphi) \in \Theta'$$

While it is desirable that interpretations preserve meaning, in some sense, there is no formal requirement that they do so. However, an informal argument that, for every $\mathscr{L}$-sentence φ, $\mathscr{I}(\varphi)$ is semantically stronger than φ may be offered as evidence that the interpretation is "natural". An interpretation $\mathscr{I}$ of Θ in Θ' is *faithful* when, for every $\mathscr{L}$-sentence φ,

$$\mathscr{I}(\varphi) \in \Theta' \implies \varphi \in \Theta$$

In many cases, interpretations will be defined in a way that allows straightforward inductive proofs to be performed. Interpretations of theory Θ in theory Θ' will often be defined by giving a *basis mapping* $\mathscr{I}$ from the parameters $\{\square, \mathbf{P}, \ldots, \mathbf{a}, \ldots\}$[6] of $\mathscr{L}$, the language of Θ, to formulas of $\mathscr{L}'$, the language of Θ', that satisfies the following three conditions:

[6] $\square$ is a special parameter of the language $\mathscr{L}$, the *universe parameter*, that does not actually

(*i*) x_0 is the only free variable of $\mathscr{I}(\square)$ — which will generally be called $\omega_{\mathscr{I}}$ rather than $\mathscr{I}(\square)$ — and

$$\Theta' \vDash \exists x_0\, \omega_{\mathscr{I}}$$

(*ii*) for every n-ary predicate $\mathbf{P}$ of $\mathscr{L}$, the free variables of $\mathscr{I}(\mathbf{P})$ are $x_0, x_1, \ldots, x_{n-1}$, and

(*iii*) for every name $\mathbf{a}$ of $\mathscr{L}$, x_0 is the only free variable of $\mathscr{I}(\mathbf{a})$ and

$$\Theta' \vDash \exists x_1\, \forall x_0\, [\, \mathscr{I}(\mathbf{a}) \leftrightarrow x_0 \equiv x_1\,].$$

These conditions can be restated as

(*i*) the formula $\omega_{\mathscr{I}}$ defines a non-empty set,

(*ii*) for every n-ary predicate $\mathbf{P}$ of $\mathscr{L}$, the formula $\mathscr{I}(\mathbf{P})$ defines an n-ary relation on the set defined by $\omega_{\mathscr{I}}$, and

(*iii*) for every name $\mathbf{a}$ of $\mathscr{L}$, the formula $\mathscr{I}(\mathbf{a})$ defines a member of the set defined by $\omega_{\mathscr{I}}$,

which emphasizes the fact that they are syntactic analogues of the three defining conditions for an $\mathscr{L}$-structure $\mathfrak{A}$,

(*i'*) the universe $|\mathfrak{A}|$ of $\mathfrak{A}$ is non-empty,

(*ii'*) for every n-ary predicate $\mathbf{P}$ of $\mathscr{L}$, $\mathbf{P}^{\mathfrak{A}}$ is an n-ary relation on $|\mathfrak{A}|$, and

(*iii'*) for every name $\mathbf{a}$ of $\mathscr{L}$, $\mathbf{a}^{\mathfrak{A}}$ is a member of $|\mathfrak{A}|$.

The mapping $\mathscr{I}$ can be extended to a map from $\mathscr{L}$-formulas to $\mathscr{L}'$-formulas in a straightforward fashion. If φ is an atomic $\mathscr{L}$-formula — say, $\mathbf{P}(\mathbf{a}, \mathbf{x})$, where $\mathbf{P}$ is a binary predicate, $\mathbf{a}$ is a name, and $\mathbf{x}$ is a variable — then $\mathscr{I}(\varphi)$ is

$$\exists \mathbf{y}\, [\, (\mathscr{I}(\mathbf{a}))(x_0/\mathbf{y}) \wedge (\mathscr{I}(\mathbf{P}))(x_0/\mathbf{y}, x_1/\mathbf{x})\,]$$

where $\mathbf{y}$ is a fresh variable. If $\mathscr{I}(\mathbf{a})$ is an equation $x_0 \equiv \mathbf{a}'$, $\mathscr{I}(\mathbf{P}(\mathbf{a}, \mathbf{x}))$ will be simplified to

$$(\mathscr{I}(\mathbf{P}))(x_0/\mathbf{a}', x_1/\mathbf{x})$$

(Note that $\mathscr{I}(\mathbf{P}(x_0, x_1, \ldots, x_{n-1}))$ is simply $\mathscr{I}(\mathbf{P})$ and that $\mathscr{I}(x_0 \equiv \mathbf{a})$ is simply $\mathscr{I}(\mathbf{a})$.) Now that $\mathscr{I}$ has been defined on all atomic formulas, it can be extended to all formulas as follows:

1. for any $\mathscr{L}$-formula φ,

$$\mathscr{I}(\neg\varphi) = \neg\, \mathscr{I}(\varphi)$$

occur in $\mathscr{L}$-formulas. This allows us to treat a structure as a mapping from the parameters of $\mathscr{L}$ to appropriate denotations — so $|\mathfrak{A}|$ is simply $\square^{\mathfrak{A}}$ — and similarly allows us to treat the basis of an interpretation as a mapping from the parameters of $\mathscr{L}$ to appropriate $\mathscr{L}'$-formulas.

2. for any $\mathscr{L}$-formulas φ and ψ,

$$\mathscr{I}(\varphi \wedge \psi) = \mathscr{I}(\varphi) \wedge \mathscr{I}(\psi)$$

3. for any $\mathscr{L}$-formulas φ and ψ,

$$\mathscr{I}(\varphi \vee \psi) = \mathscr{I}(\varphi) \vee \mathscr{I}(\psi)$$

4. for any $\mathscr{L}$-formulas φ and ψ,

$$\mathscr{I}(\varphi \rightarrow \psi) = \mathscr{I}(\varphi) \rightarrow \mathscr{I}(\psi)$$

5. there is an $\mathscr{L}'$-formula $\omega_{\mathscr{I}}$, called the *universe* of the interpretation, such that for any variable $\mathbf{x}$ and any $\mathscr{L}$-formula φ,

$$\mathscr{I}(\forall \mathbf{x}\, \varphi) = \forall \mathbf{x}\, [\omega_{\mathscr{I}}(\mathsf{x}_0/\mathbf{x}) \rightarrow \mathscr{I}(\varphi)]$$

and

$$\mathscr{I}(\exists \mathbf{x}\, \varphi) = \exists \mathbf{x}\, [\omega_{\mathscr{I}}(\mathsf{x}_0/\mathbf{x}) \wedge \mathscr{I}(\varphi)]$$

An interpretation that is defined on formulas as well as sentences and satisfies these five conditions will be said to *preserve logic.* Preservation of logic is required if straightforward inductive proofs of properties of an interpretation are to be performed. Mappings defined by starting with a basis and extending it as described above will be called *standard interpretations* of $\mathscr{L}$ in $\mathscr{L}'$.

A standard interpretation $\mathscr{I}$ of language $\mathscr{L}$ in language $\mathscr{L}'$ is an *interpretation* of $\mathscr{L}$ in $\mathscr{L}'$-theory Θ' iff

$$\Theta' \vDash \exists \mathsf{x}_0\, \omega_{\mathscr{I}}$$

and, for every individual parameter $\mathbf{a}$ of $\mathscr{L}$,

$$\Theta' \vDash \exists \mathsf{x}_1\, \forall \mathsf{x}_0\, [\, \mathscr{I}(\mathsf{x}_0 = \mathbf{a}) \leftrightarrow \mathsf{x}_0 = \mathsf{x}_1\,]$$

Of course, a standard interpretation $\mathscr{I}$ of $\mathscr{L}$ in $\mathscr{L}'$-theory Θ' is an interpretation of $\mathscr{L}$-theory Θ in Θ' if, for every sentence φ of the language of Θ,

$$\varphi \in \Theta \implies \mathscr{I}(\varphi) \in \Theta'$$

and it is faithful if, in addition, for every $\mathscr{L}$-sentence φ,

$$\mathscr{I}(\varphi) \in \Theta' \implies \varphi \in \Theta$$

The main result about standard theory interpretations that is needed for this paper is that they always preserve meaning, in a sense made precise by the following theorem.

Theorem *For any standard interpretation $\mathscr{I}$ of language $\mathscr{L}$ in $\mathscr{L}'$-theory Θ', any model $\mathfrak{A}'$ of Θ', any formula φ of $\mathscr{L}$, and any assignment x of values in $|\mathscr{I}^\partial(\mathfrak{A}')|$ to the free variables of φ,*

$$\mathfrak{A}' \vDash \mathscr{I}(\varphi)\,[x] \iff \mathscr{I}^\partial(\mathfrak{A}') \vDash \varphi\,[x]$$

Proof. It is immediate from the definition of $\mathscr{I}^\partial$ that the equivalence holds for every *atomic* formula φ of $\mathscr{L}$. That it also holds for non-atomic $\mathscr{L}$-formulas follows easily by induction. For example, if φ is $\psi \wedge \chi$, then

$$\begin{aligned}
\mathscr{I}^\partial(\mathfrak{A}') \vDash (\psi \wedge \chi)\,[x] \quad &\iff \quad \mathscr{I}^\partial(\mathfrak{A}') \vDash \psi\,[x] \text{ and } \mathscr{I}^\partial(\mathfrak{A}') \vDash \chi\,[x] \\
&\qquad\qquad \text{(Definition of } \textit{satisfaction}\text{)} \\
&\iff \quad \mathfrak{A}' \vDash \mathscr{I}(\psi)\,[x] \text{ and } \mathfrak{A}' \vDash \mathscr{I}(\chi)\,[x] \\
&\qquad\qquad \text{(Induction hypothesis)} \\
&\iff \quad \mathfrak{A}' \vDash (\mathscr{I}(\psi) \wedge \mathscr{I}(\chi))\,[x] \\
&\qquad\qquad \text{(Definition of } \textit{satisfaction}\text{)} \\
&\iff \quad \mathfrak{A}' \vDash \mathscr{I}(\psi \wedge \chi)\,[x] \\
&\qquad\qquad \text{(Definition of } \textit{interpretation}\text{)}
\end{aligned}$$

4.3 A Model-Theoretic Criterion for Theory Interpretation

Any standard interpretation $\mathscr{I}$ of the language $\mathscr{L}$ in an $\mathscr{L}'$-theory Θ' induces a "dual" mapping $\mathscr{I}^\partial$ from $\mathscr{L}'$-structures to $\mathscr{L}$-structures: roughly speaking, the denotation of a predicate of $\mathscr{L}$ is determined by the extension of the $\mathscr{L}'$-formula that interprets the predicate in the $\mathscr{L}'$-structure, and similarly for individual parameters. More formally, given a model $\mathfrak{A}' = \langle |\mathfrak{A}'|, \mathbf{P}^{\mathfrak{A}'}, \ldots, \mathbf{a}^{\mathfrak{A}'}, \ldots \rangle$ of Θ', the $\mathscr{L}$-structure $\mathscr{I}^\partial(\mathfrak{A}') = \langle |\mathscr{I}^\partial(\mathfrak{A}')|, \mathbf{P}^{\mathscr{I}^\partial(\mathfrak{A}')}, \ldots, \mathbf{a}^{\mathscr{I}^\partial(\mathfrak{A}')}, \ldots \rangle$ is determined as follows:

1. let

   $$|\mathscr{I}^\partial(\mathfrak{A}')| = \{x_0 \in |\mathfrak{A}'| : \mathfrak{A}' \vDash \omega_{\mathscr{I}}\,[x_0]\}$$

   where $\omega_{\mathscr{I}}$, the $\mathscr{L}'$-formula that bounds interpreted quantifiers, defines the subset of $|\mathfrak{A}'|$ used to interpret $|\mathfrak{A}|$,

2. for every n-ary predicate parameter $\mathbf{P}$ of $\mathscr{L}$, let

   $$\mathbf{P}^{\mathscr{I}^\partial(\mathfrak{A}')} = \{\langle x_0, x_1, \ldots, x_{n-1}\rangle \in |\mathscr{I}^\partial(\mathfrak{A}')|^n : \mathfrak{A}' \vDash \mathscr{I}(\mathbf{P}(\mathsf{x}_0, \mathsf{x}_1, \ldots, \mathsf{x}_{n-1}))\,[x_0, x_1, \ldots, x_{n-1}]\}$$

   and,

3. for every individual parameter $\mathbf{a}$ of $\mathscr{L}$, let $\mathbf{a}^{\mathscr{I}^\partial(\mathfrak{A}')}$ be the x_0 in $|\mathfrak{A}'|$ such that

$$\mathfrak{A}' \models \mathscr{I}(\mathsf{x}_0 = \mathbf{a})\,[x_0]$$

It is easy to show that if $\mathscr{I}$ is a standard interpretation of the language of the structure $\mathfrak{A}$ in the theory of the structure $\mathfrak{A}'$ and $\mathscr{I}^\partial(\mathfrak{A}')$ is isomorphic to $\mathfrak{A}$, then $\mathscr{I}$ interprets the theory of $\mathfrak{A}$ in the theory of $\mathfrak{A}'$.[7] Let f be an isomorphism from $\mathfrak{A}$ to $\mathscr{I}^\partial(\mathfrak{A}')$. Then, for every $\mathscr{L}$-formula φ and every assignment x of values in $|\mathfrak{A}|$ to the free variables of φ,

$$\begin{aligned}
\mathfrak{A} \models \varphi\,[x] \quad &\Longleftrightarrow \quad \mathscr{I}^\partial(\mathfrak{A}') \models \varphi\,[f \circ x] \\
&\qquad\qquad \text{(Definition of } \textit{isomorphism}\text{)} \\
&\Longleftrightarrow \quad \mathfrak{A}' \models \mathscr{I}(\varphi)\,[f \circ x] \\
&\qquad\qquad \text{(Theorem of Section 4.2)}
\end{aligned}$$

A fortiori, $\mathscr{I}$ is a theory interpretation.

This theorem suggests that $\mathscr{I}^\partial$ be viewed as an *abstraction mapping* that corresponds to standard interpretation $\mathscr{I}$. It says that structure $\mathfrak{A}'$ is a correct refinement of structure $\mathfrak{A}$, relative to an interpretation $\mathscr{I}$ of the language of $\mathfrak{A}$ in the language of $\mathfrak{A}'$, if $\mathscr{I}^\partial(\mathfrak{A}')$ — the result of abstracting away the features of $\mathfrak{A}'$ not expressible in the language of $\mathfrak{A}$ — is $\mathfrak{A}$ (up to isomorphism).

---

[7] Recall that the theorem that provided the basis for establishing faithfulness of standard first-order theory interpretations on our original approach was

> Interpretation $\mathscr{I}$ of the theory Θ in the theory Θ' is faithful iff, for every model $\mathfrak{A}$ of Θ, there is a model $\mathfrak{A}'$ of Θ' such that $\mathscr{I}^\partial(\mathfrak{A}')$ can be expanded to a model of the description of $\mathfrak{A}$.

or, equivalently,

> Interpretation $\mathscr{I}$ of the theory Θ in the theory Θ' is faithful iff, for every model $\mathfrak{A}$ of Θ, there is a model $\mathfrak{A}'$ of Θ' and a function from $|\mathfrak{A}|$ into $|\mathfrak{A}'|$ such that
>
> $$(\mathfrak{A}, a)_{a \in |\mathfrak{A}|} \equiv (\mathscr{I}^\partial(\mathfrak{A}'), f(a))_{a \in |\mathfrak{A}|}$$
>
> where '$\equiv$' denotes elementary equivalence.

It follows that requiring $\mathscr{I}^\partial(\mathfrak{A}')$ to be isomorphic to $\mathfrak{A}$ is not necessary for faithfulness. But this is simply an artifact of the limited expressive power of first-order logic; in ω-order logic, also known as *higher-order logic* and *the theory of types*, isomorphism is both necessary and sufficient for faithfulness. In the approach advocated in this paper, formal logical derivations have been eliminated — derivations are no longer used in proving correctness, and the question of whether a formula can be derived from the theory corresponding to a specification has been replaced by the question of whether the structure corresponding to that specification is a model of that formula — and so there is no compelling meta-theoretical reason to restrict ourselves to first-order logic. It would even be tempting to replace the notion of interpreting one *theory* in another by that of interpreting one *structure* in another ([15], pp. 212), except that we exploit intuitions about what a specification says to motivate the necessity of faithful interpretation. By focusing on *isomorphism* rather than *finite isomorphism*, the usual algebraic characterization of *elementary equivalence* [7], the dependence of this approach on a particular choice of logic is largely eliminated. However, the account below will be restricted to first-order theories of structures, to enable more direct comparison of the new approach and the original.

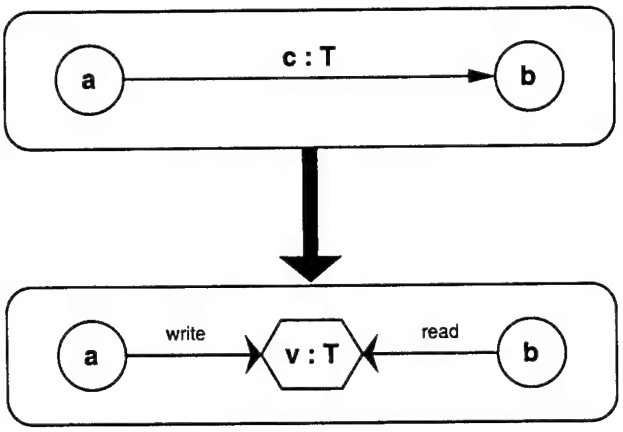

Figure 4.2: Refinement Pattern: Dataflow Channel to Variable

4.4 An Example

To enable direct comparison with the correctness proof sketch in our previous paper, the same refinement pattern, replacement of a single dataflow channel by writing and reading of a variable, will be proven correct. This pattern is presented graphically in Figure 4.2.

A refinement pattern consists of a pair of *schematic diagrams*, diagrams in which some terms have been replaced by syntactic variables. The syntactic variables of this pattern are **a**, **b**, **c**, **v**, and **T**. A pair of structures $\langle \mathfrak{D}, \mathfrak{M} \rangle$ *matches* this pattern iff, for some assignment of values to the syntactic variables, the diagram above the bold arrow depicts $\mathfrak{D}$ and the diagram below the arrow depicts $\mathfrak{M}$. The pattern is *correct* iff, for every pair of structures $\langle \mathfrak{D}, \mathfrak{M} \rangle$ that matches the pattern, $\mathfrak{M}$ is a correct refinement of $\mathfrak{D}$. So, if a pair of structures matches a correct refinement pattern, the second is a correct refinement of the first.

The class of structures depicted by the schematic diagram above the arrow

can be more formally defined as follows:

$$
\begin{aligned}
|\mathfrak{D}| &= \{a, b, c, p_o, p_i\} \cup T \\
\mathsf{Procedure}^{\mathfrak{D}} &= \{a, b\} \\
\mathsf{Channel}^{\mathfrak{D}} &= \{c\} \\
\mathsf{In\_Port}^{\mathfrak{D}} &= \{\langle p_i, b\rangle\} \\
\mathsf{Out\_Port}^{\mathfrak{D}} &= \{\langle p_o, a\rangle\} \\
\mathsf{Can\_Carry}^{\mathfrak{D}} &= \{\langle c, t\rangle : t \in T\} \\
\mathsf{Might\_Supply}^{\mathfrak{D}} &= \{\langle p_o, t\rangle : t \in T\} \\
\mathsf{Can\_Accept}^{\mathfrak{D}} &= \{\langle p_i, t\rangle : t \in T\} \\
\mathsf{Connects}^{\mathfrak{D}} &= \{\langle c, p_o, p_i\rangle\} \\
\mathbf{T}^{\mathfrak{D}} &= T \\
\mathbf{a}^{\mathfrak{D}} &= a \\
\mathbf{b}^{\mathfrak{D}} &= b \\
\mathbf{c}^{\mathfrak{D}} &= c \\
\mathbf{a\_oport}^{\mathfrak{D}} &= p_o \\
\mathbf{b\_iport}^{\mathfrak{D}} &= p_i
\end{aligned}
$$

where a, b, c, p_o, and p_i are some five distinct objects, none of which is a member of the set T.[8] Similarly, the class of structures depicted by the schematic diagram that gives the implementation of the dataflow in terms of shared memory is defined as follows:

$$
\begin{aligned}
|\mathfrak{M}| &= \{a, b, v, k_w, k_r\} \cup T \\
\mathsf{Procedure}^{\mathfrak{M}} &= \{a, b\} \\
\mathsf{Variable}^{\mathfrak{M}} &= \{v\} \\
\mathsf{Call}^{\mathfrak{M}} &= \{\langle k_w, a\rangle, \langle k_r, b\rangle\} \\
\mathsf{Can\_Hold}^{\mathfrak{M}} &= \{\langle v, t\rangle : t \in T\} \\
\mathsf{Might\_Put}^{\mathfrak{M}} &= \{\langle k_w, t\rangle : t \in T\} \\
\mathsf{Can\_Get}^{\mathfrak{M}} &= \{\langle k_r, t\rangle : t \in T\}
\end{aligned}
$$

[8]The names and types of the ports are not explicitly given by the graphical specification; the names are generated and the types are inferred, according to the principle that the type of a port is the type of the channel connected to it unless there is some explict indication to the contrary.

$$\mathsf{Writes}^{\mathfrak{M}} = \{\langle k_w, v\rangle\}$$
$$\mathsf{Reads}^{\mathfrak{M}} = \{\langle k_r, v\rangle\}$$
$$\mathbf{T}^{\mathfrak{M}} = T$$
$$\mathbf{a}^{\mathfrak{M}} = a$$
$$\mathbf{b}^{\mathfrak{M}} = b$$
$$\mathbf{v}^{\mathfrak{M}} = v$$
$$\mathbf{a\_call}^{\mathfrak{M}} = k_w$$
$$\mathbf{b\_call}^{\mathfrak{M}} = k_r$$

where a, b, v, k_w, and k_r are some five distinct objects, none of which is a member of the set T. Here, the names of the calls have been generated and the signatures of the calls inferred from types and sorts of calls.

Consider the basis mapping $\mathscr{K}$ from the parameters of the language of $\mathfrak{D}$ to formulas in the language of $\mathfrak{M}$ defined as follows:

$$\omega_{\mathscr{K}} = \mathsf{x}_0 \equiv \mathsf{x}_0$$
$$\mathscr{K}(\mathsf{Procedure}) = \mathsf{Procedure}(\mathsf{x}_0)$$
$$\mathscr{K}(\mathsf{Channel}) = \mathsf{Variable}(\mathsf{x}_0)$$
$$\mathscr{K}(\mathsf{In\_Port}) = \mathsf{Call}(\mathsf{x}_0, \mathsf{x}_1) \wedge \exists \mathsf{x}_2\, \mathsf{Can\_Get}(\mathsf{x}_0, \mathsf{x}_2)$$
$$\mathscr{K}(\mathsf{Out\_Port}) = \mathsf{Call}(\mathsf{x}_0, \mathsf{x}_1) \wedge \exists \mathsf{x}_2\, \mathsf{Might\_Put}(\mathsf{x}_0, \mathsf{x}_2)$$
$$\mathscr{K}(\mathsf{Can\_Carry}) = \mathsf{Can\_Hold}(\mathsf{x}_0, \mathsf{x}_1)$$
$$\mathscr{K}(\mathsf{Might\_Supply}) = \mathsf{Might\_Put}(\mathsf{x}_0, \mathsf{x}_1)$$
$$\mathscr{K}(\mathsf{Can\_Accept}) = \mathsf{Can\_Get}(\mathsf{x}_0, \mathsf{x}_1)$$
$$\mathscr{K}(\mathsf{Connects}() = \mathsf{Writes}(\mathsf{x}_1, \mathsf{x}_0) \wedge \mathsf{Reads}(\mathsf{x}_2, \mathsf{x}_0)$$
$$\mathscr{K}(\mathbf{T}) = \mathbf{T}(\mathsf{x}_0)$$
$$\mathscr{K}(\mathbf{a}) = \mathsf{x}_0 \equiv \mathbf{a}$$
$$\mathscr{K}(\mathbf{b}) = \mathsf{x}_0 \equiv \mathbf{b}$$
$$\mathscr{K}(\mathbf{c}) = \mathsf{x}_0 \equiv \mathbf{v}$$
$$\mathscr{K}(\mathbf{a\_oport}) = \mathsf{x}_0 \equiv \mathbf{a\_call}$$
$$\mathscr{K}(\mathbf{b\_iport}) = \mathsf{x}_0 \equiv \mathbf{b\_call}$$

The first eight clauses in the definition are determined by the general *style mapping* for interpreting dataflow style in shared memory style. The last six clauses are determined by the *identifier mapping* associated with this particular refinement step.[9] If this basis mapping is extended to an interpretation of formulas in the language of $\mathfrak{D}$ in the standard way,[10] an interpretation of the language of $\mathfrak{D}$ in the theory of $\mathfrak{M}$ results.

[9] See our previous paper [26] for a more detailed account of style and identifier mappings.
[10] See Section 4.2.

Now the value of the abstraction map $\mathscr{K}^{\mathfrak{d}}$ at $\mathfrak{M}$ can be calculated, using its definition.[11]

$$\begin{aligned}
|\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})| &= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \vDash \omega_{\mathscr{K}}[x_0]\} \\
&= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \vDash \mathsf{x}_0 = \mathsf{x}_0\ [x_0]\} \\
&= |\mathfrak{M}|
\end{aligned}$$

$$\begin{aligned}
\mathsf{Procedure}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \vDash \mathscr{K}(\mathsf{Procedure})\ [x_0]\} \\
&= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \vDash \mathsf{Procedure}(\mathsf{x}_0)\ [x_0]\} \\
&= \{a, b\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Channel}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \vDash \mathscr{K}(\mathsf{Channel})\ [x_0]\} \\
&= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \vDash \mathsf{Variable}(\mathsf{x}_0)\ [x_0]\} \\
&= \{v\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{In_Port}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathscr{K}(\mathsf{In_Port})\ [x_0, x_1]\} \\
&= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \\
&\qquad \mathfrak{M} \vDash \mathsf{Call}(\mathsf{x}_0, \mathsf{x}_1) \wedge \exists \mathsf{x}_2\, \mathsf{Can_Get}(\mathsf{x}_0, \mathsf{x}_2)\ [x_0, x_1]\} \\
&= \{\langle k_w, a\rangle\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Out_Port}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathscr{K}(\mathsf{Out_Port})\ [x_0, x_1]\} \\
&= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \\
&\qquad \mathfrak{M} \vDash \mathsf{Call}(\mathsf{x}_0, \mathsf{x}_1) \wedge \exists \mathsf{x}_2\, \mathsf{Might_Put}(\mathsf{x}_0, \mathsf{x}_2)\ [x_0, x_1]\} \\
&= \{\langle k_r, b\rangle\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Can_Carry}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathscr{K}(\mathsf{Can_Carry})\ [x_0, x_1]\} \\
&= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathsf{Can_Hold}(\mathsf{x}_0, \mathsf{x}_1)\ [x_0, x_1]\} \\
&= \{\langle c, t\rangle : t \in T\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Might_Supply}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathscr{K}(\mathsf{Might_Supply})\ [x_0, x_1]\} \\
&= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathsf{Might_Put}(\mathsf{x}_0, \mathsf{x}_1)\ [x_0, x_1]\} \\
&= \{\langle k_w, t\rangle : t \in T\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Can_Accept}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathscr{K}(\mathsf{Can_Accept})\ [x_0, x_1]\} \\
&= \{\langle x_0, x_1\rangle \in |\mathfrak{M}|^2 : \mathfrak{M} \vDash \mathsf{Can_Get}(\mathsf{x}_0, \mathsf{x}_1)\ [x_0, x_1]\} \\
&= \{\langle k_r, t\rangle : t \in T\}
\end{aligned}$$

[11] Every parameter of the language of $\mathfrak{D}$ is treated below, for the sake of completeness, but all calculations are similar and there is no need to read them all unless you are so inclined.

$$
\begin{aligned}
\mathsf{Connects}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{\langle x_0, x_1, x_2\rangle \in |\mathfrak{M}|^3 : \mathfrak{M} \models \mathscr{K}(\mathsf{Connects})\ [x_0, x_1, x_2]\} \\
&= \{\langle x_0, x_1, x_2\rangle \in |\mathfrak{M}|^3 : \\
&\qquad \mathfrak{M} \models \mathsf{Writes}(\mathsf{x}_1, \mathsf{x}_0) \wedge \mathsf{Reads}(\mathsf{x}_2, \mathsf{x}_0)\ [x_0, x_1, x_2]\} \\
&= \{\langle c, k_w, k_r\rangle\}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{T}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \models \mathscr{K}(\mathbf{T})\ [x_0]\} \\
&= \{x_0 \in |\mathfrak{M}| : \mathfrak{M} \models \mathbf{T}(\mathsf{x}_0)\ [x_0]\} \\
&= T
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{a}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathscr{K}(\mathsf{a})\ [x_0] \\
&= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathsf{x}_0 = \mathsf{a}\ [x_0] \\
&= a
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{b}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathscr{K}(\mathsf{b})\ [x_0] \\
&= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathsf{x}_0 = \mathsf{b}\ [x_0] \\
&= b
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{c}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathscr{K}(\mathsf{c})\ [x_0] \\
&= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathsf{x}_0 = \mathsf{v}\ [x_0] \\
&= v
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{a\_oport}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathscr{K}(\mathbf{a\_oport})\ [x_0] \\
&= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathsf{x}_0 = \mathbf{a\_call}\ [x_0] \\
&= k_w
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{b\_iport}^{\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})} &= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathscr{K}(\mathbf{b\_iport})\ [x_0] \\
&= (\imath x_0 \in |\mathfrak{M}|)\ \mathfrak{M} \models \mathsf{x}_0 = \mathbf{b\_call}\ [x_0] \\
&= k_r
\end{aligned}
$$

Clearly, if h from $|\mathfrak{D}|$ to $|\mathfrak{M}|$ is defined by

$$
\begin{aligned}
h(a) &= a \\
h(b) &= b \\
h(c) &= v \\
h(p_o) &= k_w \\
h(p_i) &= k_r \\
h(t) &= t \qquad \text{(for every } t \in T\text{)}
\end{aligned}
$$

then h is an isomorphism from $\mathfrak{D}$ to $\mathscr{K}^{\mathfrak{d}}(\mathfrak{M})$, and so $\mathscr{K}$ is indeed an interpretation of the theory of $\mathfrak{D}$ in the theory of $\mathfrak{M}$.

Compare this proof and the correctness proof sketch in our previous paper. In the latter, we had to show that, for any model of the dataflow theory, there was a model of the shared memory theory with a certain desirable property, viz., that the image of the shared memory model under the abstraction mapping is indistinguishable from the dataflow model using the resources of the relevant logic: see Figure 4.3. Thus, a mapping from dataflow models to shared memory models — obtained by "inverting" the equations that define the interpretation to obtain something like an inverse interpretation whose dual maps dataflow structures to shared memory structures — had to be introduced. By reducing the class of structures that correspond to a specification to a singleton, the necessity of finding an appropriate mapping from dataflow structures to shared memory structures is eliminated: see Figure 4.4. As a result, the correctness proof is reduced to straightforward calculation, requiring no creativity on the part of the prover. Correctness proof construction is, therefore, much simpler and much more straightforward when using the new approach in place of the old.

4.5 Chapter Summary

The essential difference between the two techniques stems from regarding architectural specifications as mathematical models of the systems being specified, rather than comparatively weak descriptions of those systems. Since our justification for demanding that higher-level specifications be *faithfully* interpretable in lower-level specifications was that an architectural specification should say everything that can truly be said of the system's architecture at a given level of abstraction, it is natural to demand that the logical analogues of these specifications be *complete* theories. So, this formalization of the connection between architectural specifications and logic is arguably more natural for our purposes. This is one reason for regarding the approach of this paper to be superior. Another, more practical and important, reason is that correctness proofs become much simpler, as the example shows.

It should be noted that this is a relatively minor modification of the original approach: specifications are stronger, proofs of correctness are simpler, but the basic idea of proving particular refinement steps correct by using a library of pre-verified refinement rules remains the same. Ordinary users of our architecture specification tools see no difference between the two approaches — all the refinement patterns that they are used to using are still valid — but the burden for those who wish to extend the system by validating new refinement patterns has been considerably lightened.

It should also be noted that this new approach cannot be directly applied to all conceivable refinement patterns. Some architectural descriptions cannot naturally be thought of as specifying a single architectural structure. For example, there are standard architectures, such as the X/Open Distributed Transaction Processing (DTP) architecture [46], that specify how various types of components are composed, but do not specify the number of components of each type.

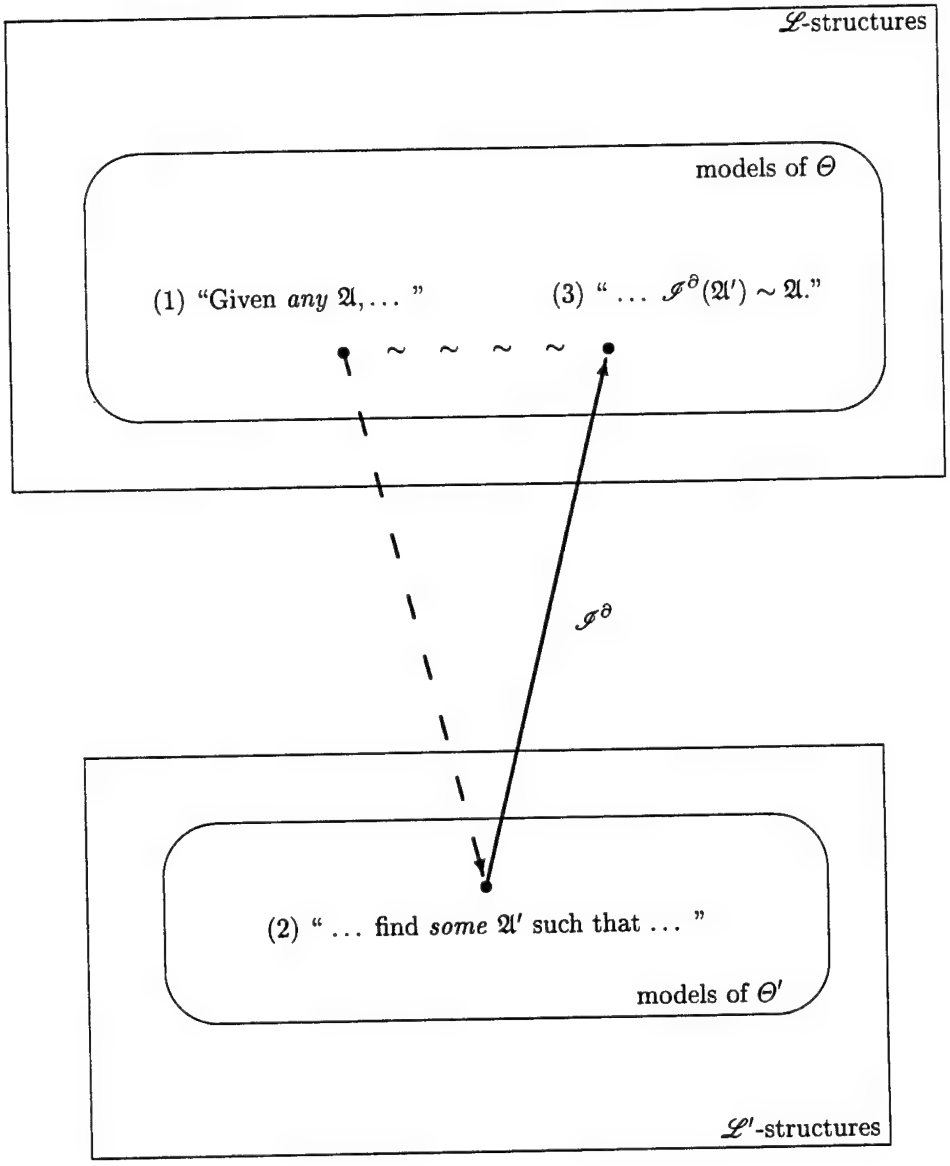

Figure 4.3: Original Proof Technique for Faithfulness

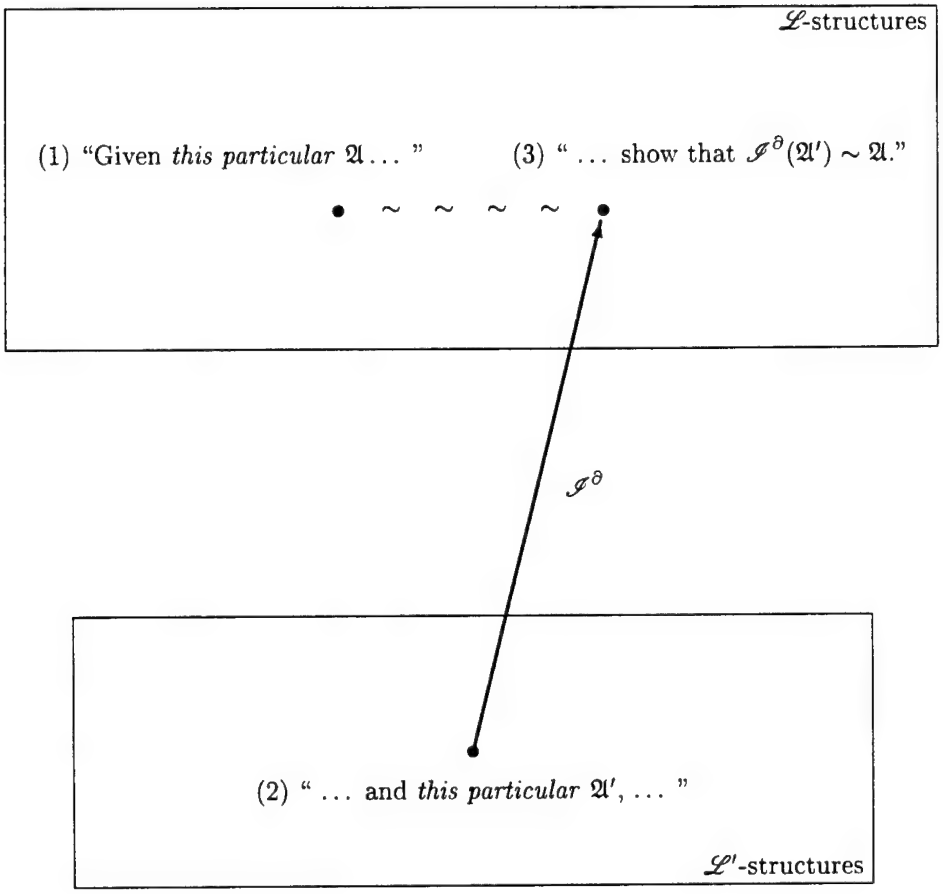

Figure 4.4: New Proof Technique for Theory Interpretation

In such cases, it is much more natural to think of the standard as specifying a class of structures. However, the new approach can be used to verify a set of simple, primitive refinement patterns that generate the refinements in a multi-level formal representation of X/Open DTP in the SADL language [28], as well as all the refinement patterns of our previous paper [26].

Chapter 5

Correctness of Architecture Transformation Rules

5.1 Introduction

In previous papers [26, 39], my colleagues and I defined what it means for a refinement step in an architecture hierarchy[1] to be *correct* and proved the correctness of a refinement pattern that says a dataflow-style architectural description consisting of two procedural components connected by a dataflow channel can be implemented as two procedural components and a variable that is written by one procedure and read by the other. This pattern is shown in Figure 5.1.

However, it should be noted that we did *not* show that a dataflow channel can *always* be replaced by a shared variable in this way. We proved that refinement step shown in Figure 5.1 is always correct; we did not prove that the refinement step shown in Figure 5.2, where the same refinement is embedded in a more complicated context, is always correct. In other words, we did not show that, if an architecture hierarchy is correct, then the extended hierarchy that results by adding a node in which a chosen dataflow channel has been refined into a read and written variable is also correct. We did not show this for a good reason: the extended hierarchy may not be correct. A very simple example illustrates this point.

The two architectural descriptions shown in Figure 5.3 differ in that the more abstract description D_1', which is an extension of description D_1 in Figure 5.1, contains a dataflow channel $\mathbf{c_2}$ that has been replaced in the more concrete description D_2' by a shared variable $\mathbf{v_2}$. According to our correctness criterion [26], every architectural description in a hierarchy should tell the *whole*

[1] An architecture hierarchy is a collection of architectural descriptions, at various levels of abstraction and often in a variety of styles, linked by refinement mappings. Since the descriptions are all intended to describe the same software architecture, they must paint a consistent portrait of that architecture. Consistency is guaranteed by the correctness of the refinement steps in the hierarchy. Thus, a hierarchy is correct iff every refinement step in it is correct.

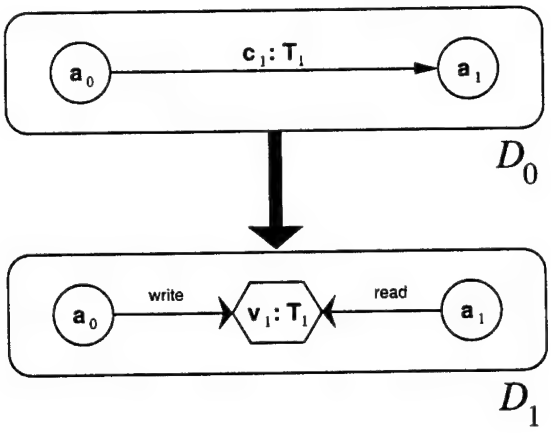

Figure 5.1: Replacing a Dataflow Channel by a Shared Variable

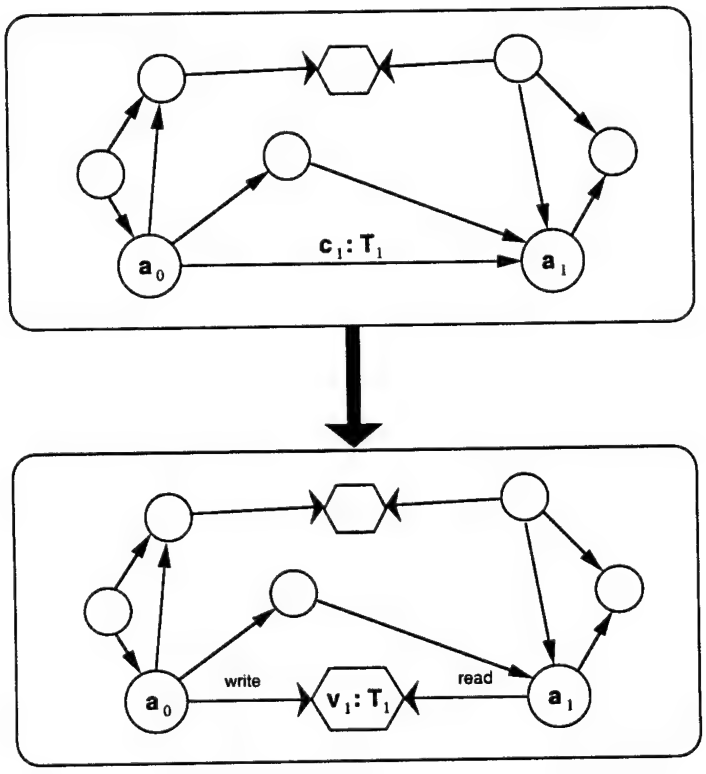

Figure 5.2: Replacing a Dataflow Channel by a Shared Variable, In Context

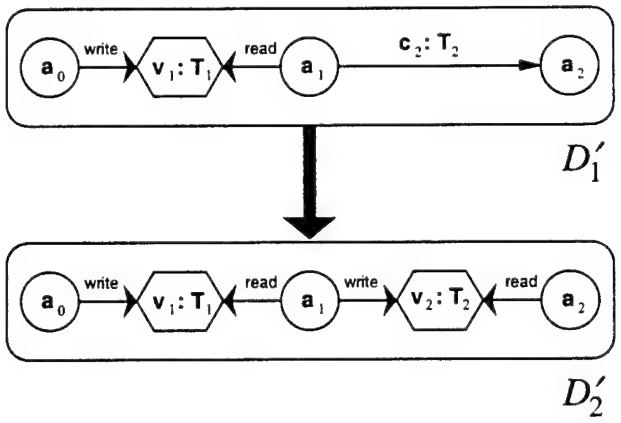

Figure 5.3: An Incorrect Hierarchy

truth — more precisely, the whole truth expressible at the level of abstraction determined by the style of the description — about the architecture. Clearly, if description D_2' in Figure 5.3 is telling the truth, then description D_1' is not telling the whole truth. The language used for D_1' has the descriptive power to say that the dataflow from $\mathbf{a_0}$ to $\mathbf{a_1}$ is implemented as a shared variable: it mentions the shared variable $\mathbf{v_2}$. In formal terms, the natural interpretation of the theory corresponding to the abstract description in the theory corresponding to the concrete description — the one that interprets dataflow in terms of shared variables, and leaves everything else "as is" — is not *faithful*.[2]

5.2 Composition versus Transformation

5.2.1 Composition

It is easy to see that the source of the problem is the mixing of dataflow-style constructs and shared-variable-style constructs in D_1'. In our previous paper [26], we employed a development methodology that avoided such non-homogeneous architectural descriptions. The basic idea was that every dataflow channel in a specification would be simultaneously replaced by its implementation. This

[2] An *interpretation* $\mathcal{I}$ of theory Θ in theory Υ is a (total) mapping from sentences in the language $\mathcal{L}$ of Θ to sentences in the language of Υ such that, for every $\mathcal{L}$-sentence φ,

$$\varphi \in \Theta \Longrightarrow \mathcal{I}(\varphi) \in \Upsilon$$

If, in addition, for every $\mathcal{L}$-sentence φ,

$$\mathcal{I}(\varphi) \in \Upsilon \Longrightarrow \varphi \in \Theta$$

then interpretation $\mathcal{I}$ is *faithful*. For more detail on interpretations, see the previous chapter on how to prove a mapping is a faithful interpretation.

complex refinement would be proven correct by showing it to be an instance of a "mega-pattern" built from verified primitive refinement patterns, using modes of composition that have been shown to preserve correctness. Figure 5.4 illustrates a simple case where a refinement of dataflow channel $\mathbf{c}_1$ that is correct in isolation and a refinement of dataflow channel $\mathbf{c}_2$ that is also correct in isolation are composed to obtain a correct refinement of the full dataflow description.

Despite the intuitive appeal of this compositional approach, there are serious difficulties in attempting to characterize the modes of composition that preserve correctness. It is instructive to see why Moriconi and Qian's attempt to work out the formal details [25, Section 8] is unsatisfactory.

First, their "definition" of $I_1 \cup I_2$ is incomplete. They write

> More precisely, if
>
> $$I_1 : \Theta_1 \to \Theta_1' \quad \text{and} \quad I_2 : \Theta_2 \to \Theta_2'$$
>
> are faithful interpretations, then we want
>
> $$I_1 \cup I_2 : \Theta_1 \cup \Theta_2 \to \Theta_1' \cup \Theta_2'$$
>
> to be a faithful interpretation. (The union of two theories is the deductive closure of the set-theoretic union of the theories.)

But how is the mapping $I_1 \cup I_2$ defined on sentences that are consequences of the union of the two theories that are not in the union of their consequences? And how is it defined on sentences that are not consequences of the union? Getting the answers to these questions right is non-trivial.[3] Without the answers, it is hard to assess the merits of their attempt in detail.

Still, it is clear that there is a fundamental difficulty with their account. What, according to Moriconi and Qian, is wrong with composing the correct refinement of Figure 5.1 with the trivially correct identity refinement on the rest of the system, to obtain the refinement shown in Figure 5.3? Simply that their second "general condition" for correct composition

> It must not be possible to infer new facts about the composite abstract architecture from the composite concrete architecture. That is, for language L_1 of Θ_1 and L_2 of Θ_2, if
>
> $$F \text{ is a sentence of } L_1 \cup L_2$$
>
> and
>
> $$\Theta_1' \cup \Theta_2' \vdash I(F)$$
>
> then we must prove that
>
> $$I(\Theta_1) \cup I(\Theta_2) \vdash I(F)$$

[3] See Section 5.4.2.

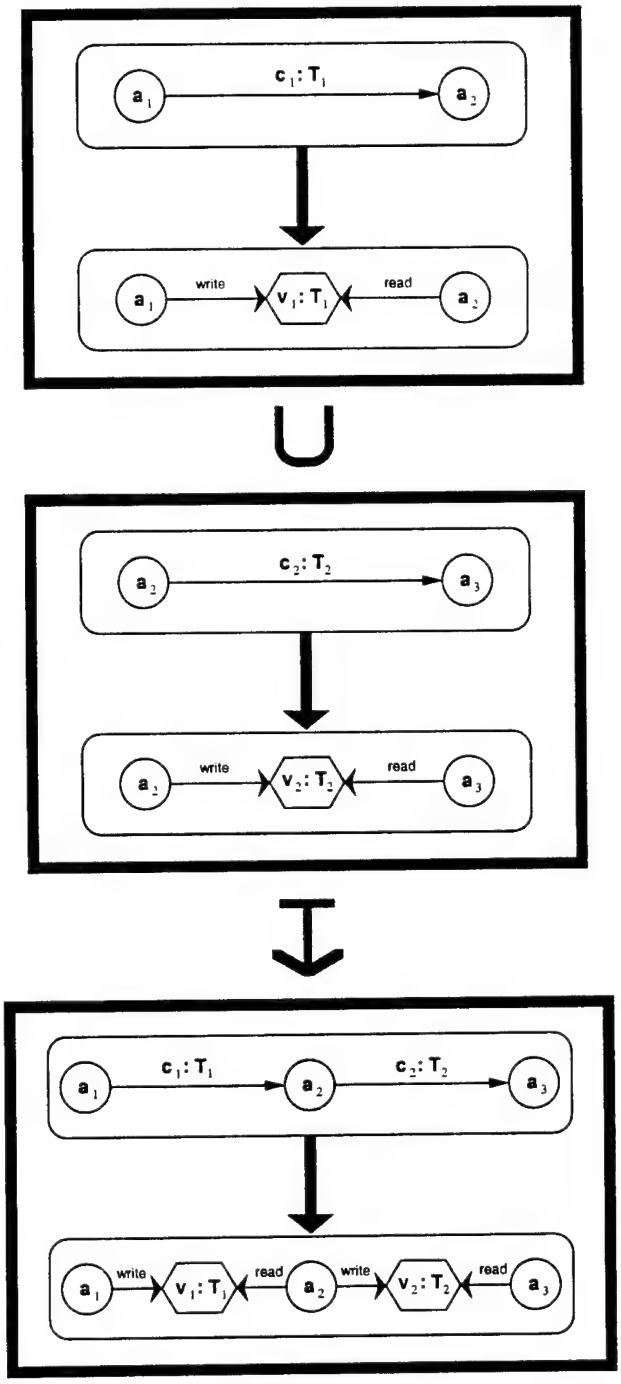

Figure 5.4: Composing Correct Refinements

is violated. But this condition is just a straightforward restatement of the requirement that the interpretation corresponding to the composite refinement is faithful. They claim that this condition is satisfied in their one specific example, which was extracted from the compiler architecture, *because* the subarchitectures "share no interface points". If they were right, the proof that a particular instance of this mode of composition preserves correctness could be reduced to a simple check that there are no common interface points in the domains of the several refinements being composed. Yet the incorrect refinement step in Figure 5.3 shows that having no shared interface points is *not* sufficient to guarantee faithfulness of this mode of composition. Moriconi and Qian thus failed to provide a single non-trivial example of a mode of composition that is guaranteed to preserve correctness. If every instance of composition requires a proof that their second "general condition" is satisfied (i.e., requires a proof of faithfulness), the fundamental motivating notion of pre-verifying the refinements and modes of composition has been abandoned.

The difficulties of finding modes of refinement composition that preserve correctness has been noted by other researchers. For example, Broy [5, pp. 3] writes

> Traditionally, compositional notions of specification and refinement are considered hard to obtain. ... Finding compositional specification methods and compositional interaction concepts [for the class of systems dealt with in this paper] is considered a difficult issue.

Of course, Broy is concerned with the usual, more liberal notion of correct refinement that consists of "adding logical properties". His behavior refinements correspond to theory extensions that are not necessarily conservative; his interface and interaction refinements correspond to theory interpretations that are not necessarily faithful. Since the faithfulness requirement we have imposed on correct structural refinement makes correctness proofs much more difficult, it should not be surprising that finding correctness-reserving modes of structural refinement composition becomes more difficult as well. So the question arises: Is there any alternative to the compositional approach that offers similar benefits?

5.2.2 Transformation

While the charms of the compositional approach are undeniable, incremental transformation offers an even more attractive alternative. The compositional approach is quite natural if the objective is to verify the correctness of a completed hierarchy. On the incremental approach, the objective is to avoid introducing mistakes during the design process, rather than catching mistakes after they have been made. Architectures are designed incrementally in practice, using an architecture design tool — perhaps a CASE tool, perhaps paper and pencil — to incrementally elaborate a hierarchy. As each design decision is made, the hierarchy is transformed to reflect the current state of the design. If all the transformations of the hierarchy preserve its correctness, and the start-

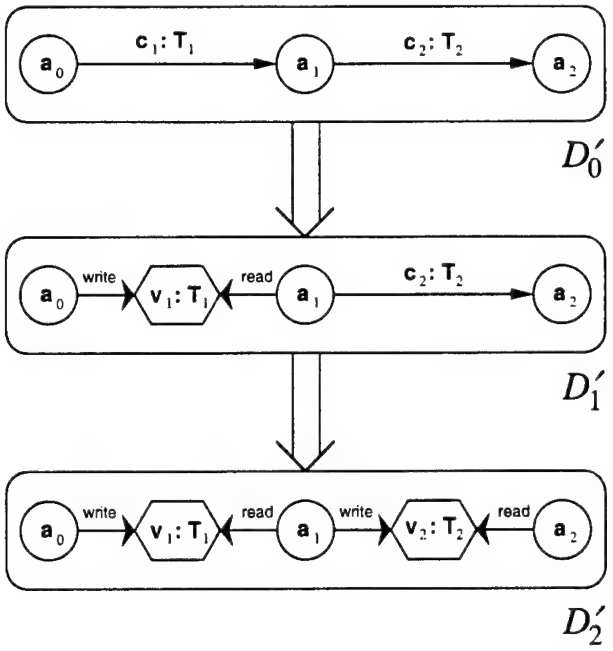

Figure 5.5: Incrementally Replacing Dataflow Channels by Shared Variables

ing point is a trivially correct refinement-free hierarchy, then the completed hierarchy that eventually results is guaranteed to be correct by construction.

But on the transformational approach, non-homogeneous specifications cannot be avoided. For example, consider an abstract architectural description containing two dataflow channels, such as D_0' of Figure 5.5. It should be possible to replace one channel by a shared variable without eliminating the option of later replacing the second by a shared variable as well. At first glance, the series of architectural transformations shown in Figure 5.5 may seem to cast doubt on our definition of correctness, for there is clearly nothing wrong with this series. What the correctness of this series actually shows is that, whatever it means to say that a transformation rule is *correct*, it cannot be the case that correct transformation is simply a matter of correct refinement. The connection between correct transformation rules and correct refinement patterns must be less direct.

5.3 From Correct Refinements to Correct Transformations

The first step in establishing the connection between correct refinement and correct transformation is to view transformations as being applied to the entire hierarchy rather than to the individual descriptions in the hierarchy. Figure 5.6 shows the same two transformations that were shown in Figure 5.5 from this perspective. Initially, the hierarchy H_0 consists of a single level. This level contains an abstract architectural description D'_0 of system dataflow. The first transformation produces a more elaborate hierarchy H_1 by introducing a second level of description, in which one of the dataflow channels of D'_0 has been replaced by a shared variable. This partly abstract, partly concrete description D'_1 is a correct refinement of D'_0, so H_1 is a correct hierarchy (i.e., every refinement step in the hierarchy is correct). The second transformation, which produces hierarchy H_2, *modifies* the lower-level description D'_1 rather than adding another level. The completed concrete-level description D'_2 is also a correct refinement of the abstract-level description D'_0, making H_2 a correct hierarchy. Note that H_2 is identical to the hierarchy produced by composition in Figure 5.4.

The original refinement pattern in Figure 5.1 thus corresponds to two transformation rules, which can be stated informally as follows.

(T_1) If an architectural hierarchy has a maximal node containing a description in pure dataflow-style, then a successor of that maximal node can be added that contains a description in which one of the dataflow channels of the maximal node's description has been refined into a shared variable.

(T_2) If an architectural hierarchy has a maximal node containing a description including a dataflow channel, and the predecessor of that node contains a description in pure dataflow-style,[4] then the maximal node's description can be replaced by a description in which the dataflow channel has been refined into a fresh shared variable.

And so the connection between correct refinement and correct transformation is now apparent:

> An architecture hierarchy transformation rule is *correct* if and only if, whenever it is applied to a correct hierarchy (i.e., a hierarchy in which every refinement step is correct), a correct hierarchy results.

In short, correct transformation rules preserve hierarchy correctness.

[4]The rationale for this restriction on the style of the predecessor will become clear when this transformation is proven correct, in Section 5.4.

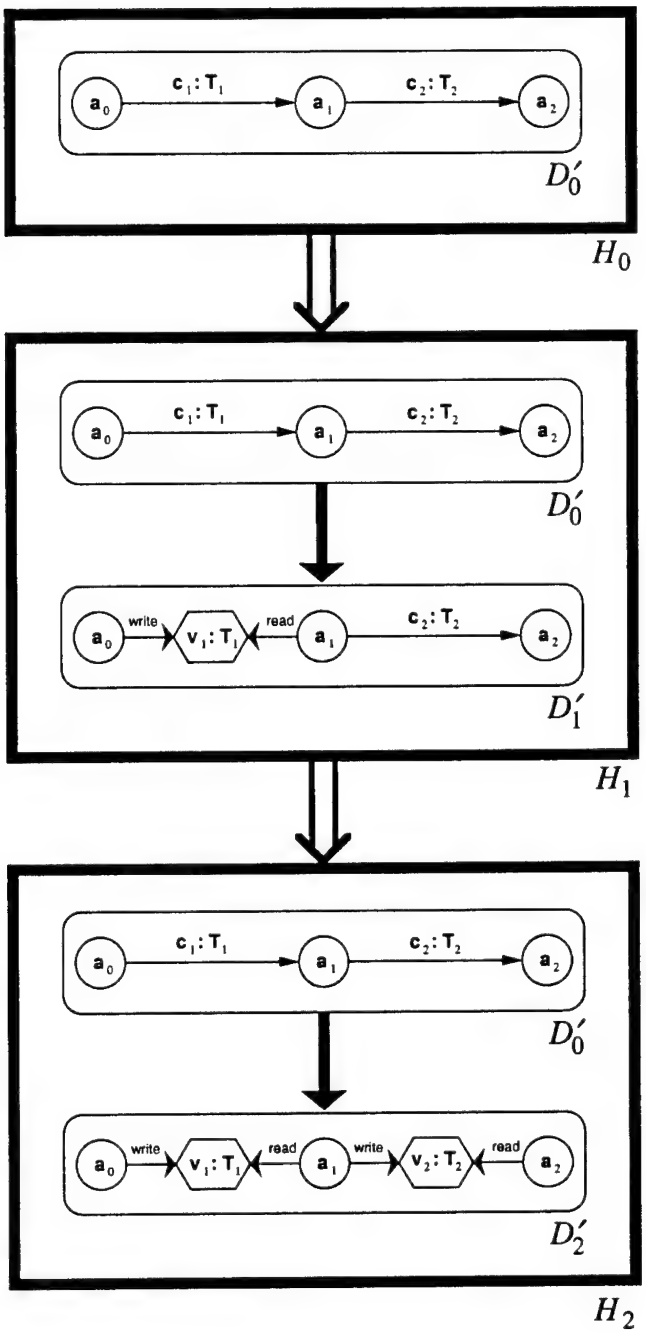

Figure 5.6: Incremental Replacement: Another Perspective

5.4 Proving Transformations Correct: two approaches

5.4.1 What more is needed to prove correctness?

Transformation T_1, the first of the two transformations above, extends a hierarchy by introducing an additional node, linked by an additional refinement step. So to say that T_1 preserves correctness of hierarchies simply means that the refinement step that is introduced is always correct. Transformation T_2 modifies an existing refinement step, so saying that it is correct means that the modification preserves correctness of the refinement step.

In a previous proof of the correctness of the refinement step in Figure 5.1 [39, pp. 7–8], the interpretation $\mathscr{K}$ determined by extending the following basis to every sentence of the language of D_0 in the standard way was shown to be a faithful interpretation of the theory of the more abstract dataflow-style architectural description D_0 in the theory of the more concrete shared-variable-style architectural description D_1.

$$
\begin{aligned}
\omega_{\mathscr{K}} &= \mathsf{x}_0 = \mathsf{x}_0 \\
\mathscr{K}(\mathsf{Procedure}) &= \mathsf{Procedure}(\mathsf{x}_0) \\
\mathscr{K}(\mathsf{Channel}) &= \mathsf{Variable}(\mathsf{x}_0) \\
\mathscr{K}(\mathsf{In\_Port}) &= \mathsf{Call}(\mathsf{x}_0, \mathsf{x}_1) \wedge \exists \mathsf{x}_2\, \mathsf{Can\_Get}(\mathsf{x}_0, \mathsf{x}_2) \\
\mathscr{K}(\mathsf{Out\_Port}) &= \mathsf{Call}(\mathsf{x}_0, \mathsf{x}_1) \wedge \exists \mathsf{x}_2\, \mathsf{Might\_Put}(\mathsf{x}_0, \mathsf{x}_2) \\
\mathscr{K}(\mathsf{Can\_Carry}) &= \mathsf{Can\_Hold}(\mathsf{x}_0, \mathsf{x}_1) \\
\mathscr{K}(\mathsf{Might\_Supply}) &= \mathsf{Might\_Put}(\mathsf{x}_0, \mathsf{x}_1) \\
\mathscr{K}(\mathsf{Can\_Accept}) &= \mathsf{Can\_Get}(\mathsf{x}_0, \mathsf{x}_1) \\
\mathscr{K}(\mathsf{Connects}) &= \mathsf{Writes}(\mathsf{x}_1, \mathsf{x}_0) \wedge \mathsf{Reads}(\mathsf{x}_2, \mathsf{x}_0) \\
\mathscr{K}(\mathbf{T}) &= \mathbf{T}(\mathsf{x}_0) \\
\mathscr{K}(\mathbf{a}_0) &= \mathsf{x}_0 = \mathbf{a}_0 \\
\mathscr{K}(\mathbf{a}_1) &= \mathsf{x}_0 = \mathbf{a}_1 \\
\mathscr{K}(\mathbf{c}_1) &= \mathsf{x}_0 = \mathbf{v}_1 \\
\mathscr{K}(\mathbf{a}_0\mathsf{\_oport}) &= \mathsf{x}_0 = \mathbf{a}_0\mathsf{\_call} \\
\mathscr{K}(\mathbf{a}_1\mathsf{\_iport}) &= \mathsf{x}_0 = \mathbf{a}_1\mathsf{\_call}
\end{aligned}
$$

Why can't the faithfulness of this interpretation be used to prove that the refinements introduced by transformation rules T_1 and T_2 are correct? It is not the interpretation that corresponds to these transformations. According to $\mathscr{K}$, *every* channel is implemented as a variable and *every* connection is implemented by reading and writing a shared variable. In incremental development, *some* connecting channels remain connecting channels.

5.4.2 First approach: modifying the interpretation

One approach to showing T_1 is correct is to modify the definition of interpretation $\mathscr{K}$, in a way that maintains faithfulness, so as to make it correspond to T_1. Ideally, the modification should preserve faithfulness in general, not just in the particular case of $\mathscr{K}$, so that proving a refinement step correct always guarantees that the transformation that extends a hierarchy by applying that refinement to a homogeneous description in a maximal node is also correct.

Unfortunately, well-known general results on extensibility of interpretations are not applicable in this case. For example, Turski and Maibaum's "Modularization Theorem" [45, pp. 174] says that an interpretation of theory Θ in theory Υ can always be extended to an interpretation of any conservative extension Θ' of Θ in a conservative extension of Υ. Despite their claim that this theorem "guarantees that an implementation of [an] extension [of an existing system] will be consistent with an implementation of the existing system," it is not applicable to the sort of "extensions" of descriptions under consideration here. Depending on details of the formalization, the theory of D_0' may or may not be an extension of D_0,[5] but it is certainly not a conservative extension. Sentences made true by the context, such as

$$\exists x_0 \, [\, \mathsf{Procedure}(x_0) \wedge \neg x_0 = \mathbf{a}_0 \wedge \neg x_0 = \mathbf{a}_1 \,]$$

will be included in the theory of D_0', but not the theory of D_0. The example Turski and Maibaum use to show that conservativeness is necessary for the theorem to hold also shows that, even if the interpretation to be extended is faithful, conservativeness remains necessary.

Since general results on extensibility do not yield the desired interpretation, a modification suited to our specific requirements must be sought. Changing the definition of $\mathscr{K}$ — thereby defining a new interpretation basis called "$[\mathbf{1} \vee \mathscr{K}]$", for reasons that the definition will make clear — so as to allow the possibility of channels being interpreted as either channels or shared variables is straightforward.

$$\begin{aligned}
\omega_{[\mathbf{1} \vee \mathscr{K}]} &= x_0 = x_0 \\
[\mathbf{1} \vee \mathscr{K}]\,(\mathsf{Procedure}) &= \mathsf{Procedure}(x_0) \\
[\mathbf{1} \vee \mathscr{K}]\,(\mathsf{Channel}) &= \mathsf{Channel}(x_0) \vee \mathsf{Variable}(x_0) \\
[\mathbf{1} \vee \mathscr{K}]\,(\mathsf{In_Port}) &= \mathsf{In_Port}(x_0, x_1) \\
&\qquad \vee \, [\, \mathsf{Call}(x_0, x_1) \wedge \exists x_2 \, \mathsf{Can_Get}(x_0, x_2) \,] \\
[\mathbf{1} \vee \mathscr{K}]\,(\mathsf{Out_Port}) &= \mathsf{Out_Port}(x_0, x_1) \\
&\qquad \vee \, [\, \mathsf{Call}(x_0, x_1) \wedge \exists x_2 \, \mathsf{Might_Put}(x_0, x_2) \,]
\end{aligned}$$

[5] If the simplified method of proving correctness is used, or the original method is used and "extremal axioms" are added to theories, sentences such as

$$\forall x_0 \, [\, \mathsf{Procedure}(x_0) \rightarrow x_0 = \mathbf{a}_0 \vee x_0 = \mathbf{a}_1 \,]$$

will be included in the theory of D_0 but not in the theory of D_0'. As a result, the theory of D_0' will not be an extension of the theory of D_0.

$$
\begin{aligned}
[\mathbf{1} \vee \mathscr{K}](\mathsf{Can\_Carry}) &= \mathsf{Can\_Carry}(x_0, x_1) \vee \mathsf{Can\_Hold}(x_0, x_1) \\
[\mathbf{1} \vee \mathscr{K}](\mathsf{Might\_Supply}) &= \mathsf{Might\_Supply}(x_0, x_1) \vee \mathsf{Might\_Put}(x_0, x_1) \\
[\mathbf{1} \vee \mathscr{K}](\mathsf{Can\_Accept}) &= \mathsf{Can\_Accept}(x_0, x_1) \vee \mathsf{Can\_Get}(x_0, x_1) \\
[\mathbf{1} \vee \mathscr{K}](\mathsf{Connects}) &= \mathsf{Connects}(x_0, x_1, x_2) \\
&\qquad \vee [\ \mathsf{Writes}(x_1, x_0) \wedge \mathsf{Reads}(x_2, x_0)\ ] \\
[\mathbf{1} \vee \mathscr{K}](\mathbf{T}) &= \mathbf{T}(x_0) \\
[\mathbf{1} \vee \mathscr{K}](\mathbf{c_1}) &= x_0 \equiv \mathbf{v_1} \\
[\mathbf{1} \vee \mathscr{K}](\mathbf{a_0}\_\mathsf{oport}) &= x_0 \equiv \mathbf{a_0}\_\mathsf{call} \\
[\mathbf{1} \vee \mathscr{K}](\mathbf{a_1}\_\mathsf{iport}) &= x_0 \equiv \mathbf{a_1}\_\mathsf{call} \\
[\mathbf{1} \vee \mathscr{K}](\mathbf{n}) &= x_0 \equiv \mathbf{n} \qquad \text{for any name } \mathbf{n} \text{ other than } \mathbf{c_1},\ \mathbf{a_0}\_\mathsf{oport}, \text{ and } \mathbf{a_1}\_\mathsf{iport}
\end{aligned}
$$

However, the resulting interpretation is not the interpretation that corresponds to transformation T_1, as an example makes clear. Consider the application of T_1 in Figure 5.6, which produces H_1 from H_0. The intended interpretation of the dataflow-language sentence

$$\mathsf{Channel}(\mathbf{c_1})$$

is

$$\mathsf{Variable}(\mathbf{v_1})$$

and the intended interpretation of the sentence

$$\mathsf{Connects}(\mathbf{c_1}, \mathbf{a_0}\_\mathsf{oport}, \mathbf{a_1}\_\mathsf{iport})$$

is

$$\mathsf{Writes}(\mathbf{a_0}\_\mathsf{call}, \mathbf{v_1}) \wedge \mathsf{Reads}(\mathbf{a_1}\_\mathsf{call}, \mathbf{v_1})$$

According to $[\mathbf{1} \vee \mathscr{K}]$, these two sentences are interpreted as

$$\mathsf{Channel}(\mathbf{v_1}) \vee \mathsf{Variable}(\mathbf{v_1})$$

and

$$\mathsf{Connects}(\mathbf{c_1}, \mathbf{a_0}\_\mathsf{oport}, \mathbf{a_1}\_\mathsf{iport}) \vee [\ \mathsf{Writes}(\mathbf{a_0}\_\mathsf{call}, \mathbf{v_1}) \wedge \mathsf{Reads}(\mathbf{a_1}\_\mathsf{call}, \mathbf{v_1})\ ]$$

Interpretation $[\mathbf{1} \vee \mathscr{K}]$ gets the general facts right — for example, it correctly interprets "there are at least two dataflow channels" as "the total number of dataflow channels and shared variables that implement dataflow is at least two" — but not the particulars.

Perhaps, then, the interpretation $[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]$ that is defined as follows on *atomic* formulas

$$[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]](\varphi) = \begin{cases} \mathscr{K}(\varphi) & \text{if } \varphi \text{ is a } \textit{sentence} \text{ in the domain of } \mathscr{K} \\ [\mathbf{1} \vee \mathscr{K}](\varphi) & \text{otherwise} \end{cases}$$

and is extended to non-atomic formulas in the standard way corresponds to T_1. (This mode of composition of interpretations — disjunction, but with basic facts treated specially — may be something like what Moriconi and Qian intended for their union operation.) While it is not entirely clear that this is the interpretation that naturally corresponds to T_1, at least there are no obvious counterexamples to that thesis.

The next step is to find the interpretation that corresponds to transformation T_2. The difference here is that T_2 is being used to modify an interpretation $\mathscr{J}$ of a pure dataflow theory in a mixed-style theory rather than introducing a new interpretation. But the basic modification being performed, replacement of a dataflow channel by a shared variable, is much the same. If $[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]$ is extended so that it applies to every sentence in the range of $\mathscr{J}$,

$$[[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]\downarrow\mathbf{1}](\varphi) = \begin{cases} [\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]](\varphi) & \text{if } [\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]] \text{ is defined for } \varphi \\ \varphi & \text{otherwise} \end{cases}$$

where the notation "$\downarrow\mathbf{1}$" is intended to suggest "defined everywhere", then T_2 can be thought of as replacing $\mathscr{J}$ by $[[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]\downarrow\mathbf{1}] \circ \mathscr{J}$, the mapping that sends the sentence φ to $[[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]\downarrow\mathbf{1}](\mathscr{J}(\varphi))$. Again, at least there are no obvious counterexamples to the proposition that $[[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]\downarrow\mathbf{1}]$ corresponds to T_2.

At this point, an attempt could be made to prove that $[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]$ is a faithful theory interpretation and that, if $\mathscr{J}$ is a faithful theory interpretation, so is $[[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]\downarrow\mathbf{1}] \circ \mathscr{J}$. But, regardless of whether the original method or the simplified method of proving faithfulness is employed, it does not seem possible to straightforwardly convert the proof that $\mathscr{K}$ is faithful into the desired proofs.[6]

This analysis suggests that there are two fundamental problems with attempts to use a calculus of interpretations as a basis for proving transformations correct. First, there is the absence of a truly compelling argument that any particular combination of interpretations naturally corresponds to a transformation. No matter how many modifications are made, there is no guarantee that yet another example that shows the interpretation fails to correspond to the transformation will not be found. Second, although the correctness of transformations seems to intuitively follow from the correctness of the refinements that suggest them, there does not seem to be any simple, uniform method of converting refinement correctness proofs into transformation correctness proofs. The examples that inspired the modifications show that the interpretations that correspond to these transformations must not be standard — the interpretation

[6]When using the original method, architectural descriptions are treated as collections of axioms, mappings are proved to be interpretations by proving the images of the axioms of the abstract theory from the axioms of the concrete theory, and faithfulness is proved by a model-theoretic method that requires "inverting" the interpretation. See our earlier paper [26] for details. When using the simplified method, architectural descriptions are taken as specifying a particular structure, mappings are proved to be theory interpretations by using the fact that a mapping of the theory of the abstract structure to the theory of the concrete structure is an interpretation if the image of the concrete structure under the abstraction map that corresponds to the mapping on sentences is isomorphic to the abstract structure, and faithfulness is automatic. See my earlier paper [39] for details.

of predicates Channel, Connects, and so on, depends on their arguments — and so neither of the techniques we have developed for proving faithfulness can be directly applied. It is not clear how either the "inverse mapping" of the original method or the abstraction mapping on models of the simplified method can be defined. In the next section, an alternative approach to proving transformation correctness that avoids both these problems will be investigated.[7]

5.4.3 Second approach: modifying the theories

The difficulties in adapting $\mathscr{K}$ to generally interpret dataflow in terms of shared variables flow from a single source: the logical theories being used to describe the architectures are too weak to distinguish between dataflow channels that get implemented as shared variables and those that do not. In terms of the example in Figure 5.6, the interpretation that T_1 introduces — call it $\mathscr{K}^+$ — should act as follows:

$$\mathscr{K}^+\big(\mathsf{Channel}(\mathbf{c}_1)\big) = \mathsf{Variable}(\mathbf{v}_1)$$
$$\mathscr{K}^+\big(\mathsf{Channel}(\mathbf{c}_2)\big) = \mathsf{Channel}(\mathbf{c}_2)$$

Since these sentences are being interpreted differently by $\mathscr{K}^+$, there is clearly some difference between the two cases that is not being reflected in the logical theory. Rather than making somewhat ad hoc modifications to $\mathscr{K}$ to account for the difference, as in the first approach, let us explore the option of modifying the logical theories.

As a first step, expand the language of dataflow $\mathscr{L}$ used to describe the more abstract architecture. Corresponding to the singulary predicate Channel, add two new singulary predicates Green_Channel and Red_Channel, and similarly for all the other predicates — In_Port, Out_Port, Can_Carry, Might_Supply, Can_Accept, and Connects — that are re-interpreted by $\mathscr{K}$. Call this expanded language $\mathscr{L}^+$.

Next, if the original correctness method is being used, extend the theory Θ of the dataflow description by adding sentences that describe the particular coloring,

$$\begin{aligned}
&\mathsf{Green_Channel}(\mathbf{c})\\
&\mathsf{Green_Connects}(\mathbf{c}, \mathbf{p}_{\mathrm{out}}, \mathbf{p}_{\mathrm{in}})\\
&\mathsf{Green_In_Port}(\mathbf{p}_{\mathrm{in}}, \mathbf{a}_{\mathrm{in}})\\
&\mathsf{Green_Out_Port}(\mathbf{p}_{\mathrm{out}}, \mathbf{a}_{\mathrm{out}})\\
&\qquad\vdots
\end{aligned}$$

where $\mathbf{c}$ is the channel that gets implemented by the application of T_1, $\mathbf{p}_{\mathrm{in}}$ and $\mathbf{p}_{\mathrm{out}}$ are the input and output ports that $\mathbf{c}$ connects, $\mathbf{a}_{\mathrm{in}}$ is the process $\mathbf{p}_{\mathrm{in}}$ is a

[7] It will turn out that $[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]$ does not correspond to T_1, and that $[[\mathscr{K}; [\mathbf{1} \vee \mathscr{K}]]\downarrow\mathbf{1}]$ does not correspond to T_2. The interpretations that correspond to these transformations do not even respect logic. So it is doubtful whether additional iterations of "find a counterexample, then modify the definition to eliminate it" would result in discovering correct definitions.

port of, and so forth, and

$$\begin{aligned}&\mathsf{Red\_Channel}(\mathbf{c})\\&\mathsf{Red\_Connects}(\mathbf{c}, \mathbf{p}_{\text{out}}, \mathbf{p}_{\text{in}})\\&\mathsf{Red\_In\_Port}(\mathbf{p}_{\text{in}}, \mathbf{a}_{\text{in}})\\&\mathsf{Red\_Out\_Port}(\mathbf{p}_{\text{out}}, \mathbf{a}_{\text{out}})\\&\quad\vdots\end{aligned}$$

for all other channels $\mathbf{c}$. The extension also adds general *color constraints* that relate colored and non-colored predicates, sentences such as

$$\forall x_0\,[\ \mathsf{Green\_Channel}(x_0) \rightarrow \mathsf{Channel}(x_0)\ ]$$

and

$$\forall x_0\,[\ \mathsf{Red\_Channel}(x_0) \rightarrow \mathsf{Channel}(x_0)\ ]$$

and also sentences such as

$$\begin{aligned}\forall x_0\,\forall x_1\,\forall x_2\,[\ &\mathsf{Green\_Connects}(x_0, x_1, x_2)\\&\leftrightarrow \mathsf{Connects}(x_0, x_1, x_2) \wedge \mathsf{Green\_Channel}(x_0)\\&\qquad \wedge \exists x_3\,\mathsf{Green\_Out\_Port}(x_1, x_3) \wedge \exists x_3\,\mathsf{Green\_In\_Port}(x_2, x_3)\ ]\end{aligned}$$

and

$$\begin{aligned}\forall x_0\,\forall x_1\,\forall x_2\,[\ &\mathsf{Red\_Connects}(x_0, x_1, x_2)\\&\leftrightarrow \mathsf{Connects}(x_0, x_1, x_2) \wedge \mathsf{Red\_Channel}(x_0)\\&\qquad \wedge \exists x_3\,\mathsf{Red\_Out\_Port}(x_1, x_3) \wedge \exists x_3\,\mathsf{Red\_In\_Port}(x_2, x_3)\ ]\end{aligned}$$

Call this extension Θ^+. Note that Θ^+ is clearly a conservative extension of Θ. A model $\mathfrak{A}$ of Θ can be expanded to a model $\mathfrak{A}^+$ of Θ^+ simply by assigning the appropriate extensions to the new predicates. For example,

$$\mathsf{Green\_Channel}^{\mathfrak{A}^+} = \{\mathbf{c}^{\mathfrak{A}^+}\}$$

and

$$\mathsf{Red\_Channel}^{\mathfrak{A}^+} = \mathsf{Channel}^{\mathfrak{A}^+} \sim \{\mathbf{c}^{\mathfrak{A}^+}\}$$

where $\mathbf{c}$ is the channel that is implemented by T_1.

Alternatively, if the simplified method is being used, just expand $\mathfrak{A}$ to $\mathfrak{A}^+$ as above, and let Θ^+ be the theory of $\mathfrak{A}^+$. (Note that this Θ^+ contains each of the sentences that was added to Θ when using the original method.) In this case too, Θ^+ is clearly a conservative extension of Θ.

Call this process of adding either Green_ or Red_ to predicates of atomic sentences to indicate how their arguments are being acted upon by the transformation *coloring*. For any $\mathscr{L}$-sentence φ, let φ^+, the *coloring of* φ, be the $\mathscr{L}^+$-sentence

$$\varphi \wedge \bigwedge_{\mathbf{n} \in N} \tau_{\mathbf{n}}$$

where N is the set of names that occur in φ and, for any name $\mathbf{n}$, the sentence $\tau_{\mathbf{n}}$ gives the most explicit typing of the denotation of $\mathbf{n}$. For example, $\tau_{\mathbf{c}}$ is Green_Channel($\mathbf{c}$) if $\mathbf{c}$ is the name of the channel being implemented by the application of T_1, and $\tau_{\mathbf{c}}$ is Red_Channel($\mathbf{c}$) if $\mathbf{c}$ is the name of any other channel.

Observation *Given the way Θ^+ has been constructed from Θ, regardless of choice of method, it is easy to see that, for every $\mathscr{L}$-sentence φ,*

$$\varphi \in \Theta \iff \varphi^+ \in \Theta^+$$

Defining a standard interpretation $\mathscr{K}_+$ of Θ^+ in the language of mixed dataflow and shared variables is now straightforward. Let $\mathbf{c}$ be the name of the channel being implemented, $\mathbf{v}$ be the name of the implementing variable, $\mathbf{p}_{\text{out}}$ be the name of the output port being implemented, $\mathbf{k}_{\text{write}}$ be the name of the implementing write call, $\mathbf{p}_{\text{in}}$ be the name of the input port being implemented, and $\mathbf{k}_{\text{read}}$ be the name of the implementing read call.

$$\begin{aligned}
\omega_{\mathscr{K}_+} &= x_0 = x_0 \\
\mathscr{K}_+(\text{Procedure}) &= \text{Procedure}(x_0) \\
\mathscr{K}_+(\text{Channel}) &= \text{Channel}(x_0) \vee \text{Variable}(x_0) \\
\mathscr{K}_+(\text{Green_Channel}) &= \text{Variable}(x_0) \\
\mathscr{K}_+(\text{Red_Channel}) &= \text{Channel}(x_0) \\
\mathscr{K}_+(\text{In_Port}) &= \text{In_Port}(x_0, x_1) \\
&\qquad \vee [\ \text{Call}(x_0, x_1) \wedge \exists x_2\, \text{Can_Get}(x_0, x_2)\ ] \\
\mathscr{K}_+(\text{Green_In_Port}) &= \text{Call}(x_0, x_1) \wedge \exists x_2\, \text{Can_Get}(x_0, x_2) \\
\mathscr{K}_+(\text{Red_In_Port}) &= \text{In_Port}(x_0, x_1) \\
\mathscr{K}_+(\text{Out_Port}) &= \text{Out_Port}(x_0, x_1) \\
&\qquad \vee [\ \text{Call}(x_0, x_1) \wedge \exists x_2\, \text{Might_Put}(x_0, x_2)\ ] \\
\mathscr{K}_+(\text{Green_Out_Port}) &= \text{Call}(x_0, x_1) \wedge \exists x_2\, \text{Might_Put}(x_0, x_2) \\
\mathscr{K}_+(\text{Red_Out_Port}) &= \text{Out_Port}(x_0, x_1) \\
\mathscr{K}_+(\text{Can_Carry}) &= \text{Can_Carry}(x_0, x_1) \vee \text{Can_Hold}(x_0, x_1) \\
\mathscr{K}_+(\text{Green_Can_Carry}) &= \text{Can_Hold}(x_0, x_1) \\
\mathscr{K}_+(\text{Red_Can_Carry}) &= \text{Can_Carry}(x_0, x_1)
\end{aligned}$$

$$
\begin{aligned}
\mathscr{K}_+(\mathsf{Might\_Supply}) &= \mathsf{Might\_Supply}(x_0, x_1) \vee \mathsf{Might\_Put}(x_0, x_1) \\
\mathscr{K}_+(\mathsf{Green\_Might\_Supply}) &= \mathsf{Might\_Put}(x_0, x_1) \\
\mathscr{K}_+(\mathsf{Red\_Might\_Supply}) &= \mathsf{Might\_Supply}(x_0, x_1) \\
\mathscr{K}_+(\mathsf{Can\_Accept}) &= \mathsf{Can\_Accept}(x_0, x_1) \vee \mathsf{Can\_Get}(x_0, x_1) \\
\mathscr{K}_+(\mathsf{Green\_Can\_Accept}) &= \mathsf{Can\_Get}(x_0, x_1) \\
\mathscr{K}_+(\mathsf{Red\_Can\_Accept}) &= \mathsf{Can\_Accept}(x_0, x_1) \\
\mathscr{K}_+(\mathsf{Connects}) &= \mathsf{Connects}(x_0, x_1, x_2) \\
&\qquad \vee [\ \mathsf{Writes}(x_1, x_0) \wedge \mathsf{Reads}(x_2, x_0)\ ] \\
\mathscr{K}_+(\mathsf{Green\_Connects}) &= \mathsf{Writes}(x_1, x_0) \wedge \mathsf{Reads}(x_2, x_0) \\
\mathscr{K}_+(\mathsf{Red\_Connects}) &= \mathsf{Connects}(x_0, x_1, x_2) \\
\mathscr{K}_+(\mathbf{T}) &= \mathbf{T}(x_0) \\
\mathscr{K}_+(\mathbf{c}) &= x_0 \equiv \mathbf{v} \\
\mathscr{K}_+(\mathbf{p}_{\mathrm{out}}) &= x_0 \equiv \mathbf{k}_{\mathrm{write}} \\
\mathscr{K}_+(\mathbf{p}_{\mathrm{in}}) &= x_0 \equiv \mathbf{k}_{\mathrm{read}} \\
\mathscr{K}_+(\mathbf{n}) &= x_0 \equiv \mathbf{n} \qquad \text{for any name } \mathbf{n} \text{ other than} \\
&\qquad\qquad\qquad\quad \mathbf{c},\ \mathbf{p}_{\mathrm{out}},\ \text{and } \mathbf{p}_{\mathrm{out}}
\end{aligned}
$$

Interpretation $\mathscr{K}^+$ can now be defined, for every $\mathscr{L}$-sentence φ, by

$$\mathscr{K}^+(\varphi) = \mathscr{K}_+(\varphi^+)$$

It should be clear that $\mathscr{K}^+$, as defined above, is the interpretation that corresponds to T_1, but consideration of an example will reinforce this point. Consider the dataflow language sentence

$$\mathsf{Connects}(\mathbf{c}, \mathbf{p}_{\mathrm{out}}, \mathbf{p}_{\mathrm{in}})$$

where $\mathbf{c}$ denotes the channel being implemented by T_1, and $\mathbf{p}_{\mathrm{out}}$ and $\mathbf{p}_{\mathrm{in}}$ denote the ports that $\mathbf{c}$ connects. The result of coloring this sentence is

$$
\begin{aligned}
&\mathsf{Connects}(\mathbf{c}, \mathbf{p}_{\mathrm{out}}, \mathbf{p}_{\mathrm{in}}) \wedge \mathsf{Green\_Channel}(\mathbf{c}) \\
&\qquad \wedge \exists x_0\, \mathsf{Green\_Out\_Port}(\mathbf{p}_{\mathrm{out}}, x_0) \wedge \exists x_0\, \mathsf{Green\_In\_Port}(\mathbf{p}_{\mathrm{in}}, x_0)
\end{aligned}
$$

Note that this sentence is equivalent, modulo the color and type constraints of $\mathscr{Y}^+$, to

$$\mathsf{Green\_Connects}(\mathbf{c}, \mathbf{p}_{\mathrm{out}}, \mathbf{p}_{\mathrm{in}})$$

Applying $\mathscr{K}_+$ to the colored sentence results in the sentence

$$
\begin{aligned}
&[\ \mathsf{Connects}(\mathbf{v}, \mathbf{k}_{\mathrm{write}}, \mathbf{k}_{\mathrm{read}}) \vee [\ \mathsf{Writes}(\mathbf{k}_{\mathrm{write}}, \mathbf{v}) \wedge \mathsf{Reads}(\mathbf{k}_{\mathrm{read}}, \mathbf{v})\ ]\ ] \\
&\qquad \wedge \mathsf{Variable}(\mathbf{v}) \\
&\qquad \wedge \exists x_0\, [\ \mathsf{Call}(\mathbf{k}_{\mathrm{write}}, x_0) \wedge \exists x_1\, \mathsf{Might\_Put}(\mathbf{k}_{\mathrm{write}}, x_0)\ ] \\
&\qquad \wedge \exists x_0\, [\ \mathsf{Call}(\mathbf{k}_{\mathrm{read}}, x_0) \wedge \exists x_1\, \mathsf{Can\_Get}(\mathbf{k}_{\mathrm{read}}, x_0)\ ]
\end{aligned}
$$

where $\mathbf{v}$ denotes the implementing variable, and $\mathbf{k}_{\text{write}}$ and $\mathbf{k}_{\text{read}}$ denote the implementing calls. This sentence is equivalent, modulo the type constraints of Υ, to

$$\mathsf{Writes}(\mathbf{k}_{\text{write}}, \mathbf{v}) \wedge \mathsf{Reads}(\mathbf{k}_{\text{read}}, \mathbf{v})$$

as desired.

5.4.4 Proving $\mathscr{K}_+$ and $\mathscr{K}^+$ faithful

Interpretation $\mathscr{K}_+$ can be shown to be a faithful interpretation of Θ^+ in the theory Υ of the mixed architecture that is introduced by the application of T_1 using essentially the same proof used to show that $\mathscr{K}$ is a faithful interpretation of the theory of the piece of the dataflow architecture that is being implemented in the theory of the implementing piece of the shared variable architecture. A sketch of the proof that $\mathscr{K}_+$ faithfully interprets the colored theory of D_0' in the theory of D_1' illustrates this point.

The structures $\mathfrak{D}_0'$ that correspond to D_0' can be formally defined as follows:

$$\begin{aligned}
|\mathfrak{D}_0'| &= \{a_0, a_1, a_2, c_1, c_2, p_0, p_1, p_2, p_3\} \cup T_1 \cup T_2 \\
\mathsf{Procedure}^{\mathfrak{D}_0'} &= \{a_0, a_1, a_2\} \\
\mathsf{Channel}^{\mathfrak{D}_0'} &= \{c_1, c_2\} \\
\mathsf{Green_Channel}^{\mathfrak{D}_0'} &= \{c_1\} \\
\mathsf{Red_Channel}^{\mathfrak{D}_0'} &= \{c_2\} \\
\mathsf{In_Port}^{\mathfrak{D}_0'} &= \{\langle p_1, a_1\rangle, \langle p_3, a_2\rangle\} \\
\mathsf{Green_In_Port}^{\mathfrak{D}_0'} &= \{\langle p_1, a_1\rangle\} \\
\mathsf{Red_In_Port}^{\mathfrak{D}_0'} &= \{\langle p_3, a_2\rangle\} \\
\mathsf{Out_Port}^{\mathfrak{D}_0'} &= \{\langle p_0, a_0\rangle, \langle p_2, a_1\rangle\} \\
\mathsf{Green_Out_Port}^{\mathfrak{D}_0'} &= \{\langle p_0, a_0\rangle\} \\
\mathsf{Red_Out_Port}^{\mathfrak{D}_0'} &= \{\langle p_2, a_1\rangle\} \\
\mathsf{Can_Carry}^{\mathfrak{D}_0'} &= \{\langle c_1, t\rangle : t \in T_1\} \cup \{\langle c_2, t\rangle : t \in T_2\} \\
\mathsf{Green_Can_Carry}^{\mathfrak{D}_0'} &= \{\langle c_1, t\rangle : t \in T_1\} \\
\mathsf{Red_Can_Carry}^{\mathfrak{D}_0'} &= \{\langle c_2, t\rangle : t \in T_2\} \\
\mathsf{Might_Supply}^{\mathfrak{D}_0'} &= \{\langle p_0, t\rangle : t \in T_1\} \cup \{\langle p_2, t\rangle : t \in T_2\} \\
\mathsf{Green_Might_Supply}^{\mathfrak{D}_0'} &= \{\langle p_0, t\rangle : t \in T_1\} \\
\mathsf{Red_Might_Supply}^{\mathfrak{D}_0'} &= \{\langle p_2, t\rangle : t \in T_2\}
\end{aligned}$$

$$
\begin{aligned}
\mathsf{Can\_Accept}^{\mathfrak{D}'_0} &= \{\langle p_1, t\rangle : t \in T_1\} \cup \{\langle p_3, t\rangle : t \in T_2\} \\
\mathsf{Green\_Can\_Accept}^{\mathfrak{D}'_0} &= \{\langle p_1, t\rangle : t \in T_1\} \\
\mathsf{Red\_Can\_Accept}^{\mathfrak{D}'_0} &= \{\langle p_3, t\rangle : t \in T_2\} \\
\mathsf{Connects}^{\mathfrak{D}'_0} &= \{\langle c_1, p_0, p_1\rangle, \langle c_2, p_2, p_3\rangle\} \\
\mathsf{Green\_Connects}^{\mathfrak{D}'_0} &= \{\langle c_1, p_0, p_1\rangle\} \\
\mathsf{Red\_Connects}^{\mathfrak{D}'_0} &= \{\langle c_2, p_2, p_3\rangle\} \\
\mathbf{T}_1^{\mathfrak{D}'_0} &= T_1 \\
\mathbf{T}_2^{\mathfrak{D}'_0} &= T_2 \\
\mathbf{a}_0^{\mathfrak{D}'_0} &= a_0 \\
\mathbf{a}_1^{\mathfrak{D}'_0} &= a_1 \\
\mathbf{a}_2^{\mathfrak{D}'_0} &= a_2 \\
\mathbf{c}_1^{\mathfrak{D}'_0} &= c_1 \\
\mathbf{c}_2^{\mathfrak{D}'_0} &= c_2 \\
\mathbf{a}_0\mathsf{\_oport}^{\mathfrak{D}'_0} &= p_0 \\
\mathbf{a}_1\mathsf{\_iport}^{\mathfrak{D}'_0} &= p_1 \\
\mathbf{a}_1\mathsf{\_oport}^{\mathfrak{D}'_0} &= p_2 \\
\mathbf{a}_2\mathsf{\_iport}^{\mathfrak{D}'_0} &= p_3
\end{aligned}
$$

where a_0, a_1, a_2, c_1, c_2, p_0, p_1, p_2, and p_3 are some nine distinct objects, none of which is a member of the set $T_1 \cup T_2$. Similarly, the structures $\mathfrak{D}'_1$ that correspond to D'_1 can be formally defined as follows:

$$
\begin{aligned}
|\mathfrak{D}'_1| &= \{a_0, a_1, a_2, v_1, c_2, k_0, k_1, p_2, p_3\} \cup T_1 \cup T_2 \\
\mathsf{Procedure}^{\mathfrak{D}'_1} &= \{a_0, a_1, a_2\} \\
\mathsf{Variable}^{\mathfrak{D}'_1} &= \{v_1\} \\
\mathsf{Channel}^{\mathfrak{D}'_1} &= \{c_2\} \\
\mathsf{Call}^{\mathfrak{D}'_1} &= \{\langle k_0, a_0\rangle, \langle k_1, a_1\rangle\}
\end{aligned}
$$

$$\begin{aligned}
\mathsf{In_Port}^{\mathfrak{D}_1'} &= \{\langle p_3, a_2\rangle\} \\
\mathsf{Out_Port}^{\mathfrak{D}_1'} &= \{\langle p_2, a_1\rangle\} \\
\mathsf{Can_Hold}^{\mathfrak{D}_1'} &= \{\langle v_1, t\rangle : t \in T_1\} \\
\mathsf{Can_Carry}^{\mathfrak{D}_1'} &= \{\langle c_2, t\rangle : t \in T_2\} \\
\mathsf{Might_Put}^{\mathfrak{D}_1'} &= \{\langle k_0, t\rangle : t \in T_1\} \\
\mathsf{Might_Supply}^{\mathfrak{D}_1'} &= \{\langle p_2, t\rangle : t \in T_2\} \\
\mathsf{Can_Get}^{\mathfrak{D}_1'} &= \{\langle k_1, t\rangle : t \in T_1\} \\
\mathsf{Can_Accept}^{\mathfrak{D}_1'} &= \{\langle p_3, t\rangle : t \in T_2\} \\
\mathsf{Writes}^{\mathfrak{D}_1'} &= \{\langle k_0, v_1\rangle\} \\
\mathsf{Reads}^{\mathfrak{D}_1'} &= \{\langle k_1, v_1\rangle\} \\
\mathbf{T}_1^{\mathfrak{D}_1'} &= T_1 \\
\mathbf{T}_2^{\mathfrak{D}_1'} &= T_2 \\
\mathbf{a}_0^{\mathfrak{D}_1'} &= a_0 \\
\mathbf{a}_1^{\mathfrak{D}_1'} &= a_1 \\
\mathbf{a}_2^{\mathfrak{D}_1'} &= a_2 \\
\mathbf{v}_1^{\mathfrak{D}_1'} &= v_1 \\
\mathbf{c}_2^{\mathfrak{D}_1'} &= c_2 \\
\mathbf{a}_0\mathsf{_call}^{\mathfrak{D}_1'} &= k_0 \\
\mathbf{a}_1\mathsf{_call}^{\mathfrak{D}_1'} &= k_1 \\
\mathbf{a}_1\mathsf{_oport}^{\mathfrak{D}_0'} &= p_2 \\
\mathbf{a}_2\mathsf{_iport}^{\mathfrak{D}_0'} &= p_3
\end{aligned}$$

where a_0, a_1, a_2, v_1, c_2, k_0, k_1, p_2, and p_3 are some nine distinct objects, none of which is a member of the set $T_1 \cup T_2$.

The calculation of the value of the abstraction map $\mathscr{K}_+^{\mathfrak{D}}$ at $\mathfrak{D}_1'$ is very much like the calculation of $\mathscr{K}^{\mathfrak{D}}$ at $\mathfrak{D}_1$: the Green_ predicates correspond to the predicates of the language of D_0, the Red_ predicates are the identity, and the non-colored predicates are simply unions. I will show the calculation for just one

triple of predicates, since they are all so similar.

$$\begin{aligned}
\mathsf{Channel}^{\mathscr{K}_+^{\mathcal{O}}(\mathfrak{D}_1')} &= \{x_0 \in |\mathfrak{D}_1'| : \mathfrak{D}_1' \vDash \mathscr{K}_+(\mathsf{Channel})[x_0]\} \\
&= \{x_0 \in |\mathfrak{D}_1'| : \mathfrak{D}_1' \vDash \mathsf{Channel}(\mathsf{x}_0) \vee \mathsf{Variable}(\mathsf{x}_0)[x_0]\} \\
&= \{v_1, c_2\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Green_Channel}^{\mathscr{K}_+^{\mathcal{O}}(\mathfrak{D}_1')} &= \{x_0 \in |\mathfrak{D}_1'| : \mathfrak{D}_1' \vDash \mathscr{K}_+(\mathsf{Channel})[x_0]\} \\
&= \{x_0 \in |\mathfrak{D}_1'| : \mathfrak{D}_1' \vDash \mathsf{Variable}(\mathsf{x}_0)[x_0]\} \\
&= \{v_1\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{Red_Channel}^{\mathscr{K}_+^{\mathcal{O}}(\mathfrak{D}_1')} &= \{x_0 \in |\mathfrak{D}_1'| : \mathfrak{D}_1' \vDash \mathscr{K}_+(\mathsf{Channel})[x_0]\} \\
&= \{x_0 \in |\mathfrak{D}_1'| : \mathfrak{D}_1' \vDash \mathsf{Channel}(\mathsf{x}_0)[x_0]\} \\
&= \{c_2\}
\end{aligned}$$

And, just as in the proof of $\mathscr{K}$, it is easy to see that the function h from $|\mathfrak{D}_0'|$ to $|\mathscr{K}_+^{\mathcal{O}}(\mathfrak{D}_1')|$ defined by

$$\begin{aligned}
h(a_0) &= a_0 \\
h(a_1) &= a_1 \\
h(a_2) &= a_2 \\
h(c_1) &= v_1 \\
h(c_2) &= c_2 \\
h(p_0) &= k_0 \\
h(p_1) &= k_1 \\
h(p_2) &= p_2 \\
h(p_3) &= p_3 \\
h(t) &= t \qquad \text{for every } t \in T_1 \cup T_2
\end{aligned}$$

is an isomorphism. Hence, $\mathscr{K}_+$ is faithful.

That $\mathscr{K}^+$ is a faithful interpretation of Θ in Υ is an immediate corollary: for every $\mathscr{L}$-sentence φ,

$$\begin{aligned}
\varphi \in \Theta &\iff \varphi^+ \in \Theta^+ && \text{(Observation 1)} \\
&\iff \mathscr{K}_+(\varphi^+) \in \Upsilon && (\mathscr{K}_+ \text{ is faithful}) \\
&\iff \mathscr{K}^+(\varphi) \in \Upsilon && \text{(Definition of } \mathscr{K}^+)
\end{aligned}$$

This completes the proof that T_1 is correct. In contrast to the earlier attempt to modify $\mathscr{K}$ to deal with examples correctly, there can be little doubt that $\mathscr{K}^+$ corresponds to T_1: it was defined to be the result of applying $\mathscr{K}$ to a designated piece of the dataflow architecture — the piece that was "colored green" — and leaving the rest alone. In addition, note that the bulk of the correctness proof consisted of a proof that $\mathscr{K}_+$ is faithful, which is a straightforward modification of the proof that $\mathscr{K}$ is faithful.

5.4.5 "... and similarly for T_2"

The idea of expanding the theory and then using $\mathscr{K}$ to interpret it worked so well for T_1 that trying to do the same for T_2 is the natural course of action. The first step, expanding the language of the more abstract description by adding the predicates Green_Channel, Red_Channel, Green_Connects, Red_Connects, and so on, is exactly the same. The only difference in this case is that the language $\mathscr{L}$ that is being expanded is not just the pure language of dataflow plus names of the objects that make up the particular architecture. It also contains some additional vocabulary introduced by the earlier implementation transformations. For example, it may contain the language of shared variables as a result of an earlier application of T_1 or T_2. The next step — conservatively extending the $\mathscr{L}$-theory Θ of the more abstract description to obtain $\mathscr{L}^+$-theory Θ^+ — is, again, exactly the same for T_2 as for T_1. And, of course, sentences can be colored and Observation 1 continues to hold, just as in the case of T_1.

But, in this case, the theory Υ of the more concrete description is analogously extended.[8] Just as the channel in the abstract description that the transformation will implement must be distinguished from those that are left alone, the new variable that is introduced in the concrete-level description must be distinguished from those that were already present. Again, coloring will be employed, this time using Green_ for the freshly introduced variable — since it corresponds to the "green" channel — and Yellow_ for the pre-existing variables, and similarly for other types. Call the extended theory $\Upsilon^\times$.

Now define an interpretation $\mathscr{K}_\times$ of Θ^+ in $\Upsilon^\times$ by slightly modifying and extending the definition of $\mathscr{K}_+$.

$$
\begin{aligned}
\omega_{\mathscr{K}_\times} &= x_0 \equiv x_0 \\
\mathscr{K}_\times(\mathsf{Procedure}) &= \mathsf{Procedure}(x_0) \\
\mathscr{K}_\times(\mathsf{Channel}) &= \mathsf{Channel}(x_0) \vee \mathsf{Variable}(x_0) \\
\mathscr{K}_\times(\mathsf{Green\_Channel}) &= \mathsf{Green\_Variable}(x_0) \\
\mathscr{K}_\times(\mathsf{Red\_Channel}) &= \mathsf{Channel}(x_0) \\
\mathscr{K}_\times(\mathsf{Variable}) &= \mathsf{Yellow\_Variable}(x_0) \\
\mathscr{K}_\times(\mathsf{In\_Port}) &= \mathsf{In\_Port}(x_0, x_1) \\
&\qquad \vee [\ \mathsf{Call}(x_0, x_1) \wedge \exists x_2\ \mathsf{Can\_Get}(x_0, x_2)\ ] \\
\mathscr{K}_\times(\mathsf{Green\_In\_Port}) &= \mathsf{Green\_Call}(x_0, x_1) \wedge \exists x_2\ \mathsf{Green\_Can\_Get}(x_0, x_2) \\
\mathscr{K}_\times(\mathsf{Red\_In\_Port}) &= \mathsf{In\_Port}(x_0, x_1) \\
\mathscr{K}_\times(\mathsf{Call}) &= \mathsf{Yellow\_Call}(x_0, x_1) \\
\mathscr{K}_\times(\mathsf{Can\_Get}) &= \mathsf{Yellow\_Can\_Get}(x_0, x_1)
\end{aligned}
$$

[8] The necessity of coloring Υ will be demonstrated below.

$$\mathscr{K}_\times(\mathsf{Out\_Port}) = \mathsf{Out\_Port}(x_0, x_1) \vee [\ \mathsf{Call}(x_0, x_1) \wedge \exists x_2\ \mathsf{Might\_Put}(x_0, x_2)\ ]$$
$$\mathscr{K}_\times(\mathsf{Green\_Out\_Port}) = \mathsf{Green\_Call}(x_0, x_1) \wedge \exists x_2\ \mathsf{Green\_Might\_Put}(x_0, x_2)$$
$$\mathscr{K}_\times(\mathsf{Red\_Out\_Port}) = \mathsf{Out\_Port}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Might\_Put}) = \mathsf{Yellow\_Might\_Put}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Can\_Carry}) = \mathsf{Can\_Carry}(x_0, x_1) \vee \mathsf{Can\_Hold}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Green\_Can\_Carry}) = \mathsf{Green\_Can\_Hold}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Red\_Can\_Carry}) = \mathsf{Can\_Carry}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Can\_Hold}) = \mathsf{Yellow\_Can\_Hold}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Might\_Supply}) = \mathsf{Might\_Supply}(x_0, x_1) \vee \mathsf{Might\_Put}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Green\_Might\_Supply}) = \mathsf{Green\_Might\_Put}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Red\_Might\_Supply}) = \mathsf{Might\_Supply}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Might\_Put}) = \mathsf{Yellow\_Might\_Put}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Can\_Accept}) = \mathsf{Can\_Accept}(x_0, x_1) \vee \mathsf{Can\_Get}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Green\_Can\_Accept}) = \mathsf{Green\_Can\_Get}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Red\_Can\_Accept}) = \mathsf{Can\_Accept}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Can\_Get}) = \mathsf{Yellow\_Can\_Get}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Connects}) = \mathsf{Connects}(x_0, x_1, x_2) \vee [\ \mathsf{Writes}(x_1, x_0) \wedge \mathsf{Reads}(x_2, x_0)\ ]$$
$$\mathscr{K}_\times(\mathsf{Green\_Connects}) = \mathsf{Green\_Writes}(x_1, x_0) \wedge \mathsf{Green\_Reads}(x_2, x_0)$$
$$\mathscr{K}_\times(\mathsf{Red\_Connects}) = \mathsf{Connects}(x_0, x_1, x_2)$$
$$\mathscr{K}_\times(\mathsf{Writes}) = \mathsf{Yellow\_Writes}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathsf{Reads}) = \mathsf{Yellow\_Reads}(x_0, x_1)$$
$$\mathscr{K}_\times(\mathbf{T}) = \mathbf{T}(x_0)$$
$$\mathscr{K}_\times(\mathbf{P}) = \mathbf{P}(x_0, x_1, \ldots, x_{n-1}) \quad \text{for every other predicate } \mathbf{P} \text{ of } \mathscr{L}, \text{ where } \mathbf{P} \text{ is } n\text{-ary}$$
$$\mathscr{K}_\times(\mathbf{c}) = x_0 = \mathbf{v}$$
$$\mathscr{K}_\times(\mathbf{p}_{\mathrm{out}}) = x_0 = \mathbf{k}_{\mathrm{write}}$$
$$\mathscr{K}_\times(\mathbf{p}_{\mathrm{in}}) = x_0 = \mathbf{k}_{\mathrm{read}}$$
$$\mathscr{K}_\times(\mathbf{n}) = x_0 = \mathbf{n} \quad \text{for any name } \mathbf{n} \text{ of } \mathscr{L} \text{ other than } \mathbf{c},\ \mathbf{p}_{\mathrm{out}},\ \text{and } \mathbf{p}_{\mathrm{out}}$$

Showing that $\mathscr{K}_\times$ faithfully interprets the colored theory of D_1' in the colored theory of D_2' is very much like showing that $\mathscr{K}_+$ faithfully interprets the colored

theory of D_0' in the theory of D_1'. For example,

$$\begin{aligned}\mathsf{Channel}^{\mathscr{K}_\times^\partial(\mathfrak{D}_2')} &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathscr{K}_+(\mathsf{Channel})\,[x_0]\} \\ &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathsf{Channel}(\mathsf{x}_0) \vee \mathsf{Variable}(\mathsf{x}_0)\,[x_0]\} \\ &= \{v_1, v_2\}\end{aligned}$$

$$\begin{aligned}\mathsf{Green\_Channel}^{\mathscr{K}_\times^\partial(\mathfrak{D}_2')} &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathscr{K}_+(\mathsf{Green\_Channel})\,[x_0]\} \\ &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathsf{Green\_Variable}(\mathsf{x}_0)\,[x_0]\} \\ &= \{v_2\}\end{aligned}$$

$$\begin{aligned}\mathsf{Red\_Channel}^{\mathscr{K}_\times^\partial(\mathfrak{D}_2')} &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathscr{K}_+(\mathsf{Red\_Channel})\,[x_0]\} \\ &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathsf{Channel}(\mathsf{x}_0)\,[x_0]\} \\ &= \emptyset\end{aligned}$$

$$\begin{aligned}\mathsf{Variable}^{\mathscr{K}_\times^\partial(\mathfrak{D}_2')} &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathscr{K}_+(\mathsf{Variable})\,[x_0]\} \\ &= \{x_0 \in |\mathfrak{D}_2'| : \mathfrak{D}_2' \vDash \mathsf{Yellow\_Variable}(\mathsf{x}_0)\,[x_0]\} \\ &= \{v_1\}\end{aligned}$$

and so $\mathfrak{D}_1'$ and $\mathscr{K}_\times^\partial(\mathfrak{D}_2')$ are indeed isomorphic. But note that if Υ had not been colored, the proof would have broken down. The extension of Green_Channel in the image of $\mathfrak{D}_2'$ under the abstraction map $\mathscr{K}_\times^\partial$ would have been the set of all the shared variables of $\mathfrak{D}_2'$ (i.e., $\{v_1, v_2\}$) rather than just the set of "green" ones, $\{v_2\}$. Similarly, the extension of Variable in the image of $\mathfrak{D}_2'$ under the abstraction map would have been $\{v_1, v_2\}$ rather than just $\{v_1\}$.

All that remains to complete the proof of T_2's correctness is to define the interpretation $\mathscr{K}^\times$ of Θ in Υ and prove that, if $\mathscr{J}$ is a faithful interpretation of some pure dataflow theory Σ in Θ, which was produced by composing some number of interpretations that correspond to transformations, then $\mathscr{K}^\times \circ \mathscr{J}$ is a faithful interpretation of Σ in Υ. The pattern of our earlier example cannot be slavishly followed, and $\mathscr{K}^\times(\varphi)$ defined as $\mathscr{K}_\times(\varphi^+)$, because this sentence is in the language of $\Upsilon^\times$, not the language of Υ. Apparently, what is needed is some way of "uncoloring" $\mathscr{K}_\times(\varphi^+)$. The same sort of examples that showed coloring Υ was necessary for faithfully interpreting Θ^+ show that $\Upsilon^\times$ cannot naturally be faithfully interpreted in Υ, i.e., that there is no *general* method for faithfully uncoloring sentences in the language of $\Upsilon^\times$. In fact, even a simple sentence of the form

$$\neg\mathsf{Yellow\_Variable}(\mathbf{a})$$

might be impossible to uncolor, since there are two ways **a** can fail to be a "yellow" channel, by being a non-channel and by being a "green" channel. But "yellow" and "green" channels are indistinguishable except with respect to color, so simply changing Yellow_Variable to either Variable or Channel without regard

to a will violate faithfulness. Fortunately, faithfully uncoloring *every* sentence in the language of Υ is not required. A restatement of the correctness criterion for $\mathscr{K}^{\times}$ will make this point clearer.

For any set Δ of sentences in the language of Θ, and any mapping $\mathscr{I}$ of the sentences in the language of Θ to sentences in the language of Υ, $\mathscr{I}$ will be said to be a *Δ-faithful interpretation of Θ in Υ* if and only if, for every sentence φ in Δ,

$$\varphi \in \Theta \iff \mathscr{I}(\varphi) \in \Upsilon$$

The usual notion of faithfulness is a special case in which Δ is the set of all sentences in the language of Θ.

It follows directly from the definition of Δ-faithfulness that the following two conditions on faithful interpretation $\mathscr{J}$ of theory Σ in theory Θ and interpretation $\mathscr{I}$ of theory Θ in theory Υ are equivalent.

1. $\mathscr{I} \circ \mathscr{J}$ is a faithful interpretation of theory Σ in theory Υ.
2. $\mathscr{I}$ is a $\mathscr{J}[\Gamma]$-faithful interpretation of theory Θ in theory Υ, where Γ is the set of all sentences in the language of Σ.

So it is not necessary to faithfully uncolor *every* sentence in the language of $\Upsilon^{\times}$, only those sentences that are $\mathscr{K}_{\times}((\mathscr{J}(\psi))^{+})$ for some dataflow language sentence ψ. It seems likely that coloring cannot affect whether a sentence $\mathscr{K}_{\times}((\mathscr{J}(\psi))^{+})$ is a consequence of $\Upsilon^{\times}$: it has been shown that, for every sentence ψ in the language of Σ,

$$\begin{aligned}
\psi \in \Sigma &\iff \mathscr{J}(\psi) \in \Theta && (\mathscr{J} \text{ is faithful}) \\
&\iff (\mathscr{J}(\psi))^{+} \in \Theta^{+} && (\text{Observation 1}) \\
&\iff \mathscr{K}_{\times}\left((\mathscr{J}(\psi))^{+}\right) \in \Upsilon^{\times} && (\mathscr{K}_{\times} \text{ is faithful})
\end{aligned}$$

so whether or not a sentence mentioning "yellow" variables, "green" variables, and so on, is a consequence of $\Upsilon^{\times}$ is entirely determined by whether some other sentence that only talks about channels and connections is a member of Σ.

This intuition can be confirmed. For any sentence φ in the language of $\Upsilon^{\times}$, let φ^{-}, the *uncoloring* of φ, be the sentence in the language of Υ that results when all colored predicates are replaced by their uncolored counterparts. For example, both the predicate Green_Variable and the predicate Yellow_Variable are replaced by the predicate Variable. By showing that, for every sentence ψ in the language of Σ,

$$\mathscr{K}_{\times}\left((\mathscr{J}(\psi))^{+}\right) \in \Upsilon^{\times} \iff \left(\mathscr{K}_{\times}\left((\mathscr{J}(\psi))^{+}\right)\right)^{-} \in \Upsilon$$

the proof that T_2 is correct, with its corresponding $\mathscr{J}[\Gamma]$-faithful interpretation $\mathscr{K}^{\times}$ defined by

$$\mathscr{K}^{\times}(\varphi) = (\mathscr{K}_{\times}(\varphi^{+}))^{-}$$

will be complete.

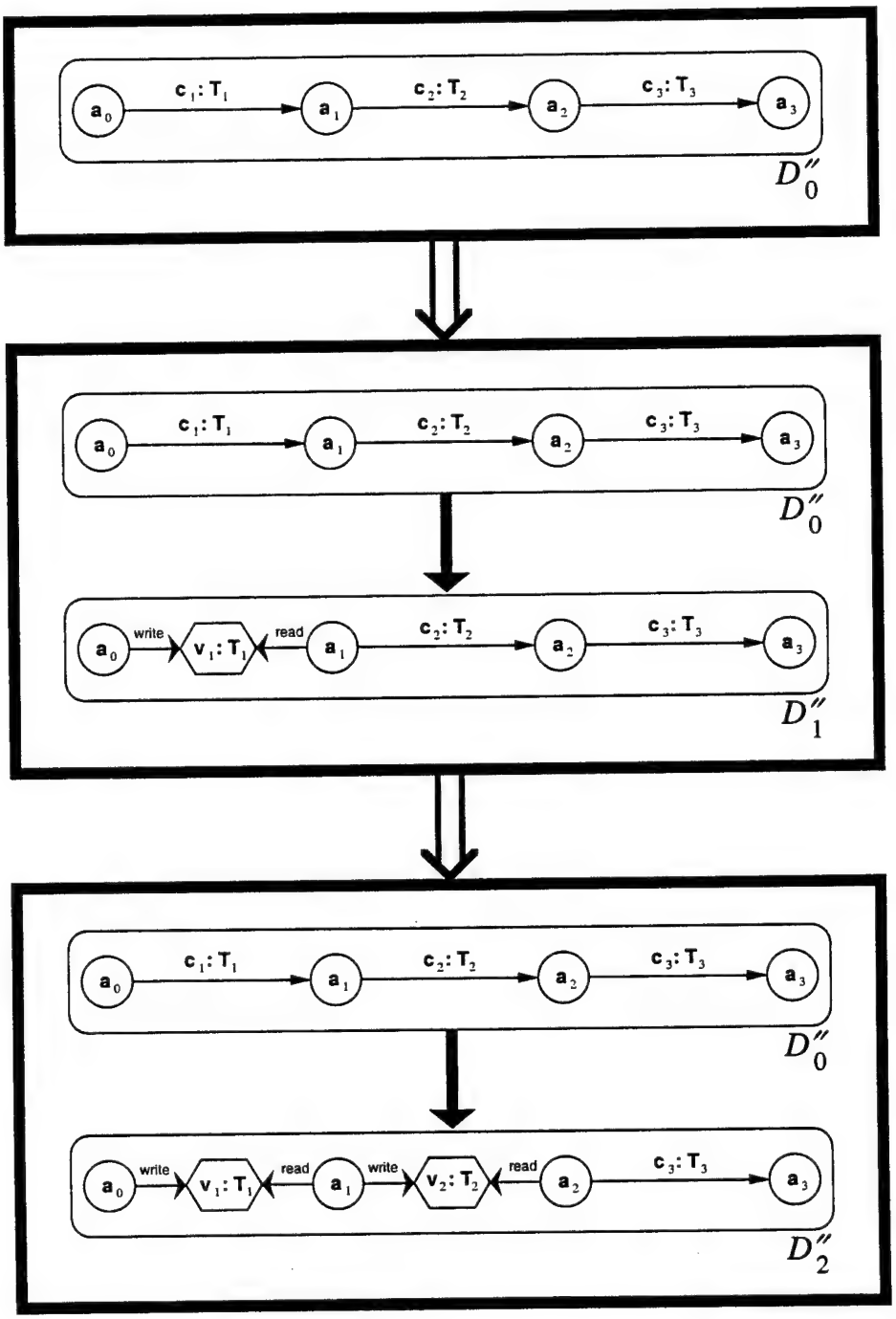

Figure 5.7: Incremental Replacement with Additional Channel

To make the structure of the proof a bit clearer, an example, based on Figure 5.7 — which is similar to the familiar transformation of hierarchies in Figure 5.6, but with an additional channel that remains unimplemented — will be examined first. Consider a sentence of the form

$$\mathsf{Channel}(\mathbf{n})$$

where name $\mathbf{n}$ is a name that denotes a value in the domain of $\mathfrak{D}_0''$, the mathematical structure that corresponds to description D_0''. There are four cases to be considered.

Case 1: $\mathbf{n}$ *denotes the channel that is implemented by the application of* T_1.

In this case, coloring the sentence for the application of T_1 produces

$$\mathsf{Channel}(\mathbf{n}) \wedge \mathsf{Green\_Channel}(\mathbf{n})$$

which, according to the color constraints of the expanded theory of $\mathfrak{D}_0''$, is equivalent to

$$\mathsf{Green\_Channel}(\mathbf{n})$$

Applying $\mathscr{K}_+$ to the colored sentence results in

$$[\,\mathsf{Channel}(\mathbf{m}) \vee \mathsf{Variable}(\mathbf{m})\,] \wedge \mathsf{Variable}(\mathbf{m})$$

where name $\mathbf{m}$ denotes the implementing variable, which is equivalent, according to the type constraints of the theory of $\mathfrak{D}_1''$, to

$$\mathsf{Variable}(\mathbf{m})$$

Coloring this sentence for the application of T_2 has no effect, since it only adds a logically redundant conjunct. So applying $\mathscr{K}_\times$ yields a sentence equivalent to

$$\mathsf{Yellow\_Variable}(\mathbf{m})$$

which can be uncolored to yield

$$\mathsf{Variable}(\mathbf{m})$$

Case 2: $\mathbf{n}$ *denotes the channel that is implemented by the application of* T_2.

Coloring the sentence for the application of T_1 produces

$$\mathsf{Channel}(\mathbf{n}) \wedge \mathsf{Red\_Channel}(\mathbf{n})$$

which, according to the color constraints of the expanded theory of $\mathfrak{D}_0''$, is equivalent to

$$\mathsf{Red\_Channel}(\mathbf{n})$$

Applying $\mathscr{K}_+$ to the colored sentence results in

$$[\,\mathsf{Channel}(\mathbf{n}) \vee \mathsf{Variable}(\mathbf{n})\,] \wedge \mathsf{Channel}(\mathbf{n})$$

which is equivalent, according to the type constraints of the theory of $\mathfrak{D}_1''$, to

$$\mathsf{Channel}(\mathbf{n})$$

Coloring this sentence for the application of T_2 results in

$$\mathsf{Channel}(\mathbf{n}) \wedge \mathsf{Green\_Channel}(\mathbf{n})$$

which is equivalent modulo the color constraints of the expanded theory of $\mathfrak{D}_1''$ to

$$\mathsf{Green\_Channel}(\mathbf{n})$$

Applying $\mathscr{K}_\times$ yields

$$[\,\mathsf{Channel}(\mathbf{m}) \vee \mathsf{Variable}(\mathbf{m})\,] \wedge \mathsf{Green\_Variable}(\mathbf{m})$$

where $\mathbf{m}$ denotes the implementing variable, which is equivalent modulo the color constraints of the expanded theory of $\mathfrak{D}_2''$ to

$$\mathsf{Green\_Variable}(\mathbf{m})$$

Uncoloring produces

$$[\,\mathsf{Channel}(\mathbf{m}) \vee \mathsf{Variable}(\mathbf{m})\,] \wedge \mathsf{Variable}(\mathbf{m})$$

which is equivalent modulo the type constraints of the theory of $\mathfrak{D}_2''$ to

$$\mathsf{Variable}(\mathbf{m})$$

Case 3: $\mathbf{n}$ *denotes the channel that remains unimplemented.*
Coloring the sentence for the application of T_1 again produces

$$\mathsf{Channel}(\mathbf{n}) \wedge \mathsf{Red\_Channel}(\mathbf{n})$$

or, equivalently,

$$\mathsf{Red\_Channel}(\mathbf{n})$$

and applying $\mathscr{K}_+$ to the colored sentence results in

$$[\,\mathsf{Channel}(\mathbf{n}) \vee \mathsf{Variable}(\mathbf{n})\,] \wedge \mathsf{Channel}(\mathbf{n})$$

or

$$\mathsf{Channel}(\mathbf{n})$$

just as in Case 2. Coloring for the application of T_2 also results in

$$\mathsf{Channel}(\mathbf{n}) \wedge \mathsf{Red\_Channel}(\mathbf{n})$$

or

$$\mathsf{Red\_Channel}(\mathbf{n})$$

Applying $\mathscr{K}_\times$ yields

$$[\,\mathsf{Channel}(\mathbf{n}) \vee \mathsf{Variable}(\mathbf{n})\,] \wedge \mathsf{Channel}(\mathbf{n})$$

or

$$\mathsf{Channel}(\mathbf{n})$$

Uncoloring is thus unnecessary.

Case 4: $\mathbf{n}$ *denotes a non-channel.*

In this case, the first coloring produces

$$\mathsf{Channel}(\mathbf{n}) \wedge \tau_{\mathbf{n}}$$

which is equivalent, modulo the color constraints of the expanded theory of $\mathfrak{D}_0''$, to a contradiction. Subsequent applications of $\mathscr{K}_+$, the second coloring, $\mathscr{K}_\times$, and the uncoloring all preserve this property of being equivalent, modulo general color and type constraints, to a contradiction.

In any case, uncoloring a sentence of the form

$$\mathsf{Channel}(\mathbf{n})$$

in $\Upsilon^\times$ yields a sentence in Υ, and uncoloring a sentence of this form not in $\Upsilon^\times$ yields a sentence not in Υ. This occurs because, at every stage, the sentence is either mapped to its corresponding implementing sentence or simply left alone if it is true, and is immediately eliminated as a candidate for inclusion in any implementing theory if it is false.

The proof that the same is true when the interpretation $\mathscr{K}^+$ is replaced by another interpretation $\mathscr{J}$ determined by transformations and the predicate Channel is replaced by another singulary predicate $\mathbf{P}$ is quite similar.

Case 1: $\mathbf{n}$ *has been implemented as* $\mathbf{m}$ *by an earlier transformation.*

Coloring $\mathscr{J}(\mathbf{P}(\mathbf{n}))$ for the application of T_2 adds conjuncts that are implied by Θ, such as $\tau_{\mathbf{m}}$, applying $\mathscr{K}_\times$ might add Yellow_ to some of the predicates of $\mathscr{J}(\mathbf{P}(\mathbf{n}))$ and some of the other conjuncts, but then uncoloring removes Yellow_ leaving $\mathscr{J}(\mathbf{P}(\mathbf{n}))$ conjoined with predicates that are implied by Υ. So,

$$\mathscr{J}(\mathbf{P}(\mathbf{n})) \in \Theta \iff (\mathscr{K}_\times((\mathscr{J}(\mathbf{P}(\mathbf{n})))^+))^- \in \Upsilon$$

in this case.

Case 2: $\mathbf{n}$ *denotes the channel that is implemented as variable* $\mathbf{m}$ *by the application of* T_2.

Coloring this sentence for the application of T_2 results in

$$\mathbf{P(n)} \wedge \mathsf{Green\_Channel(n)}$$

which is equivalent modulo the color constraints of the expanded theory of $\mathfrak{D}_1''$ to

$$\mathsf{Green\_P(n)}$$

Applying $\mathscr{K}_\times$ yields

$$\mathsf{Green\_}(\mathscr{K}(\mathbf{P(n)})) \wedge \mathsf{Green\_Variable(m)}$$

where $\mathsf{Green\_}(\mathscr{K}(\mathbf{P(n)}))$ is the result of replacing every predicate $\mathbf{Q}$ of $\mathscr{K}(\mathbf{P(n)})$ for which a predicate $\mathsf{Green\_Q}$ has been introduced by $\mathsf{Green\_Q}$. Uncoloring produces

$$\mathscr{K}(\mathbf{P(n)}) \wedge \mathsf{Variable(m)}$$

where the second conjunct is implied by Υ. So, by faithfulness of the original $\mathscr{K}$,

$$\mathscr{J}(\mathbf{P(n)}) \in \Theta \iff (\mathscr{K}_\times((\mathscr{J}(\mathbf{P(n)}))^+))^- \in \Upsilon$$

in this case as well.

Case 3: $\mathbf{n}$ *denotes a channel that remains unimplemented.*

Coloring for the application of T_2 also results in

$$\mathbf{P(n)} \wedge \mathsf{Red\_Channel(n)}$$

or

$$\mathsf{Red\_P(n)}$$

Applying $\mathscr{K}_\times$ yields

$$\mathscr{K}(\mathbf{P(n)}) \wedge \mathsf{Channel(n)}$$

Uncoloring is again unnecessary, and so once again

$$\mathscr{J}(\mathbf{P(n)}) \in \Theta \iff (\mathscr{K}_\times((\mathscr{J}(\mathbf{P(n)}))^+))^- \in \Upsilon$$

Case 4: $\mathbf{n}$ *denotes a non-channel.*

Either the type constraints of Σ require that the argument of $\mathbf{P}$ denotes a channel, or they forbid that the argument of $\mathbf{P}$ denotes a channel. If the former, there is equivalence to a contradiction after coloring, which is maintained by $\mathscr{K}_\times$ and uncoloring. If the latter, then $\mathscr{J}(\mathbf{P(n)}$ is unaffected

by the application of T_2, and it will be a member of Υ iff it is a member of Θ. So, in this last case, as in all the others,

$$\mathscr{J}(\mathbf{P(n)}) \in \Theta \iff (\mathscr{K}_{\times}((\mathscr{J}(\mathbf{P(n)}))^{+}))^{-} \in \Upsilon$$

Essentially the same argument works for multinary predicates, and ultimately for sentences in general,[9] but the notation becomes more complicated and the number of cases to be considered increases. Ignoring details, it pretty much boils down to the fact that coloring has no effect on membership in the theories that correspond to the descriptions, because the conjuncts that are added due to coloring are all implied by the relevant theories, and so $\mathscr{K}^{\times}$ either implements according to $\mathscr{K}$ (which is faithful) or doesn't do anything (which is also faithful). Since this was our a priori reason for thinking T_2 is correct, the proof can be considered a formalization of that intuition.

Just as in the case of the proof of the correctness of T_1, this proof has several advantages over a proof produced by ad hoc modification of $\mathscr{K}$. First, there is no doubt that interpretation $\mathscr{K}^{\times}$ corresponds to T_2, since it was defined as applying $\mathscr{K}$ to the appropriately colored piece of the architecture. Second, the correctness of T_2 followed from the faithfulness of $\mathscr{K}$ in straightforward, albeit tedious, fashion.

5.5 The Main Results

Re-examining the definitions of $\mathscr{K}_{+}$ and $\mathscr{K}_{\times}$, the proof that $\mathscr{K}_{+}$ is faithful, and the proof that, if $\mathscr{J}$ is faithful, then so is $\mathscr{K}_{\times} \circ \mathscr{J}$, will reveal that the same constructions could be used with another interpretation $\mathscr{I}$ in place of $\mathscr{K}$, provided $\mathscr{I}$ corresponds to some simple implementation pattern[10] in the same way that $\mathscr{K}$ corresponds to the implementation pattern in Figure 5.1. Thus, what has actually been shown by the arguments above is

Theorem *If*

1. *$\mathscr{I}$ is the faithful interpretation that corresponds to a correct simple implementation pattern in the same way that $\mathscr{K}$ corresponds to the refinement pattern in Figure 5.1,*

2. *the architectural description in a maximal node in a correct architecture hierarchy contains a subarchitecture that matches the abstract description of the pattern, and*

3. *the entire architectural description of the maximal node is in the same style as the abstract description of the pattern,*

[9] More precisely, induction on formulas is employed. Variables are treated by assigning them values, formulas containing free variables are interpreted as if the variables were the names of those values. By generalizing over all assignments, the induction is completed.

[10] Intuitively, a simple implementation pattern is one that says a connector or component of some type can always be replaced by some structure composed of components and connectors.

then the transformation rule that extends the hierarchy by adding a successor to the maximal node containing a description in which the subarchitecture is replaced by the concrete description of the refinement pattern (and all else remains unaltered) is correct, with corresponding interpretation $\mathscr{I}^+$.

and

Theorem *If*

1. $\mathscr{I}$ *is the faithful interpretation that corresponds to a correct simple implementation pattern in the same way that* $\mathscr{K}$ *corresponds to the refinement pattern in Figure 5.1,*

2. *the architectural description in a maximal node in a refinement hierarchy contains a subarchitecture that matches the abstract description of the pattern,*

3. *the predecessor of the maximal node contains a description with the same subarchitecture, and*

4. *the entire architectural description of the predecessor is in the same style as the abstract description of the pattern,*

then the transformation rule that modifies the hierarchy by replacing the description in the maximal node by a description in which the concrete description of the refinement pattern is substituted for the subarchitecture (and all else remains unaltered) is correct, with corresponding interpretation $\mathscr{I}^\times$.

This pair of results reduces the problem of verifying the pair of transformations that corresponds to a simple implementation pattern to verifying the pattern "in isolation", a problem that has already been shown to be quite feasible. The fundamental problems with the compositional approach — uncertainty whether the composite interpretation actually corresponds to the transformation and having to prove that a variety of modes of interpretation composition preserve faithfulness — have been completely avoided.

5.6 Increasing Formality

A question that the statement of the results and the arguments supporting them immediately raises is: Why are both the results and the arguments stated so informally? The reason is that they rely on an intuitive notion of what a refinement pattern is. The particular patterns used as illustrations were presented by drawing pictures of them, but these pictures have no precise semantics and so the arguments relied on an informal, intuitive understanding of what they meant.

In SADL, the formal architectural description language we have developed [28], refinements can be thought of as having the form

An architectural description that matches pattern Π
and satisfies constraints Δ
can be refined to a description that matches pattern Π'
and satisfies constraints Δ'.

For example, one of our earlier papers [26, Pattern 3] contains the following refinement for merging shared variables: an architectural description that matches the pattern[11]

```
@a0: Procedure[@@p00 -> @@p01]
@a1: Procedure[@@p10 -> @@p11]
@a2: Procedure[@@p20 -> @@p21]
@v1: Variable[@t]
@v2: Variable[@t]
@c0: ASSERTION = Writes(@a0, @v1)
@c1: ASSERTION = Reads(@a1, @v1)
@c2: ASSERTION = Writes(@a1, @v2)
@c3: ASSERTION = Reads(@a2, @v2)
@@rest
```

can be refined to a description that matches the pattern

```
@a0: Procedure[@@p00 -> @@p01]
@a1: Procedure[@@p10 -> @@p11]
@a2: Procedure[@@p20 -> @@p21]
@v:  Variable[@t]
@c0: ASSERTION = Writes(@a0, @v)
@c1: ASSERTION = Reads(@a1, @v)
@c2: ASSERTION = Writes(@a1, @v)
@c3: ASSERTION = Reads(@a2, @v)
@@rest
```

provided certain additional constraints on the architectural descriptions are satisfied.

One reason this pattern cannot be applied without requiring satisfaction of additional constraints is that collapsing the two variables into one might introduce an additional dataflow path from procedure $\mathbf{a}_0$ to procedure $\mathbf{a}_2$. However, this additional dataflow will not occur if, whenever $\mathbf{v}_1$ is written by $\mathbf{a}_0$, $\mathbf{v}_2$ is written by $\mathbf{a}_1$ before $\mathbf{v}_2$ is read by $\mathbf{a}_2$. A sufficient condition that guarantees this temporal property is that the procedures are run sequentially: first $\mathbf{a}_0$, next $\mathbf{a}_1$, and finally $\mathbf{a}_2$.[12] One way of sequentially adding this constraint to the refinement[13] is to include it as part of the concrete pattern, as follows.

[11]The notation is essentially self-explanatory once you know that terms that begin with "@" are pattern variables, the SADL-equivalent of the metavariables in logical formulas, and that pattern variables beginning with "@@" are matched with collections of subtrees of the abstract syntax tree rather than individual subtrees.

[12]Note that this assumes that Writes means *does write*, not *can write*.

[13]Explicitly adding constraints at the concrete level is quite appropriate if the refinement

```
@a0: Procedure[@@p00 -> @@p01]
@a1: Procedure[@@p10 -> @@p11]
@a2: Procedure[@@p20 -> @@p21]
@v:  Variable[@t]
@c0: ASSERTION = Writes(@a0, @v)
@c1: ASSERTION = Reads(@a1, @v)
@c2: ASSERTION = Writes(@a1, @v)
@c3: ASSERTION = Reads(@a2, @v)
@c4: ASSERTION = Starts_After_Finish_Of(@a1, @a0)
@c5; ASSERTION = Starts_After_Finish_Of(@a2, @a1)
@@rest
```

With this modification, it is easy to prove that this refinement is correct *in isolation*, i.e., when `@@rest` is matched to the empty set in both patterns.

But it is easy to see that further restrictions are necessary when the pattern is applied in context. The problem is that procedures other than $\mathbf{a}_0$, $\mathbf{a}_1$, and $\mathbf{a}_2$ may be reading or writing $\mathbf{v}_1$ or $\mathbf{v}_2$ — e.g., one of the two variables may have been introduced by collapsing two other variables — and so collapsing $\mathbf{v}_1$ and $\mathbf{v}_2$ may introduce additional dataflow paths involving those other procedures. One or more additional constraints, say

$$\forall x\,[\ \mathsf{Procedure}(x) \wedge \mathsf{Writes}(x, \mathbf{v}_1) \rightarrow x \equiv \mathbf{a}_0\ ]$$
$$\forall x\,[\ \mathsf{Procedure}(x) \wedge \mathsf{Reads}(x, \mathbf{v}_1) \rightarrow x \equiv \mathbf{a}_1\ ]$$
$$\forall x\,[\ \mathsf{Procedure}(x) \wedge \mathsf{Writes}(x, \mathbf{v}_2) \rightarrow x \equiv \mathbf{a}_1\ ]$$
$$\forall x\,[\ \mathsf{Procedure}(x) \wedge \mathsf{Reads}(x, \mathbf{v}_2) \rightarrow x \equiv \mathbf{a}_2\ ]$$

must be true of the abstract description to ensure that no additional dataflow will be introduced by the variable collapse.[14] It can now be shown that the refinement is correct, in the following sense: if an architectural description matches the first pattern above, implies the four additional conditions listed above, and is consistent with the order-of-execution constraints added to the second pattern, then it can be correctly refined to a description that matches the modified second pattern.

This example illustrates the sort of complexities that can arise in verifying refinement patterns more complicated than the simple implementation patterns considered in this paper. In particular, it shows that, in order to turn Results 1 and 2 into real theorems, a full formalization of refinements will be required, including a precise definition of *simple implementation pattern*, and that, as a result, the statement of the theorems is liable to be somewhat more complex than the informal statement of the results of this paper might suggest. If the ultimate goal is to automatically generate correct transformations from *any* refinement pattern that has been proven correct in isolation, some manner

is being used to generate a more concrete description from the abstract description, but is less appropriate if the objective is to check the correctness of existing hierarchies where the constraints may merely be implied by some other explicit constraints.

[14] These conditions are merely sufficient for correctness, not necessary.

of automatically generating the additional conditions that must be satisfied is needed.

Chapter 6

Checking Correctness of Refinement Steps

6.1 Motivation

The process of specifying an architecture often begins by providing a very high-level description of it. This description characterizes the architecture in terms of a few abstract components, perhaps the principal functions the system must perform and some data stores. These components are linked by abstract connectors, perhaps indicating dataflow or control flow relationships among the components. This abstract description provides an easily understood overview of the entire system architecture, but omits so much detail that it provides relatively little guidance to someone charged with implementing the architecture using programming-language-level and operating-system-level constructs. So the abstract description must be successively refined — with complex components and connectors decomposed into simpler parts, and abstract specifications of operations and relationships replaced by more concrete specifications — until an appropriate amount of detail has been added. It usually is desirable to continue the refinement until implementation-level constructs have replaced all the abstractions.

Alternatively, architecting a system can consist of assembling instances of reusable component and connector types selected from a library. Such libraries effectively make implementation-level more abstract, and reduce the conceptual gap between the requirements specification and the implemented architecture. Nevertheless, combining a large number of components and connectors in complex ways can easily result in an architecture that is hard to understand and analyze. So, it is desirable to generate more easily comprehensible abstract representations of the implementation-level architecture.

In either case, the end product of the architecting process is typically a collection of architectural descriptions, at different levels of abstraction and often in different styles [10]. The more abstract descriptions are linked to the more

concrete descriptions by interpretation mappings. An interpretation mapping says how the abstractions are implemented.[1] It sends each sentence in the language of the abstract description to a corresponding sentence in the language of the concrete description. For example, the fact that some component a is implemented by components $\mathsf{a}_1, \mathsf{a}_2, \ldots, \mathsf{a}_n$ would be indicated by mapping the sentence

$$\mathsf{Component}(\mathsf{a})$$

to the sentence

$$\mathsf{Component}(\mathsf{a}_1) \wedge \mathsf{Component}(\mathsf{a}_2) \wedge \ldots \wedge \mathsf{Component}(\mathsf{a}_n)$$

The collection of architectural descriptions and interpretation mappings that compose the complete architectural specification is called an *architecture hierarchy.*

There are many advantages to formalizing refinement and abstraction in system development: a library of refinement or abstraction transformations provides a "corporate knowledge base" of standard, or preferred, development patterns; mechanizing the application of these transformations lessens the likelihood of clerical errors during the development process; reuse of the transformations will result in greater validation of the patterns they codify, and so on. But one of the most fundamental advantages of formalization is that it allows the average developer to produce abstraction hierarchies that are guaranteed to be consistent. In other words, the use of verified transformations in the development process will guarantee that abstractions accurately characterize implementations, albeit more abstractly. A verified refinement transformation is one that has been proven to produce a correct implementation of whatever it is applied to. A verified abstraction transformation is one that has been proven to produce a correct abstraction of whatever it is applied to.

Even when attention is restricted to the case of architectures, there is some debate as to exactly what *correct* should mean. We have proposed a somewhat stricter-than-usual criterion for correctness [26], while others have argued that the standard criterion is preferable [35]. For present purposes, any reasonable criterion will do perfectly well. By a reasonable criterion, I mean one that characterizes correctness in terms of preservation of consequences. The standard correctness criterion is that every consequence of the abstract description must be a consequence of the concrete description as well. More precisely, for every sentence φ in the language of the abstract description,

$$\Theta \models \varphi \quad \Longrightarrow \quad \Theta' \models \mathscr{I}(\varphi)$$

where Θ is the logical theory that formalizes the abstract description, Θ' is the theory that formalizes the concrete description, and $\mathscr{I}$ is the interpretation

[1] For more details on characterizing implementation steps using interpretation mappings, see our earlier paper [26].

mapping that links the two theories.[2] A mapping $\mathcal{I}$ that satisfies this condition is called an *interpretation of* Θ *in* Θ'. Our proposed stronger criterion replaces the conditional with a biconditional, i.e., requires that the interpretation mapping be a *faithful theory interpretation.* One might also employ weaker-than-standard criteria, where only some consequences of the theory — properties of special interest — need be preserved.

What all these criteria have in common is that they justify the use of formal reasoning about the architecture based on the more abstract descriptions. If some sentence is shown to be a formal consequence of the abstract architectural theory, the concrete theory is known to correctly implement the abstract theory, and the sentence is among those that the correctness criterion guarantees are preserved by the implementation, then the sentence is known to be a consequence of the concrete theory as well. It is correctness guarantees that link the results of abstract analyses to the real world.

The usual approach to producing a correctness guarantee is restricting the architect to the use of verified transformations. This approach suffers from a problem, in practice. Even given a fairly mature library of verified transformations, it would hardly be surprising if an architect found himself unable to perform a certain refinement or abstraction step that he believed to be correct because the required transformation has not been included in the library. Expecting the typical system architect to produce a formal proof that the step is correct is unrealistic, yet the presence of a single unverified implementation step in the hierarchy voids the correctness guarantee provided by the restriction to verified transformations. Is there any way to allow the user to include such arbitrary steps in the development of the architecture hierarchy, while maintaining a correctness guarantee?

6.2 Proof-carrying Architectures

Our answer to this question is based on the notion of checking the correctness of steps in architecture hierarchy development. By *checking*, I mean automatically performing some calculation that shows the step is correct. Checking can be substantially simpler than verification, because it is focused on a particular step. Verifying a transformation means showing that it *always* produces correct results, while checking a transformation step means showing that a correct result was obtained *in one specific case.* Thus, checking entirely avoids the sometimes difficult problem of characterizing the preconditions required for the transformation to produce correct results [41].

Our intital approach to checking transformation steps was inspired work by George Necula and Peter Lee on compilers that generate proof-carrying code (PCC) [30]. Their basic idea is, rather than attempting to prove that the transformations performed by a compiler always produce code with certain desired

[2] Our earlier paper [26] explains how to formalize architectural descriptions as logical theories. Since architectural description languages are largely declarative, the process is quite straightforward.

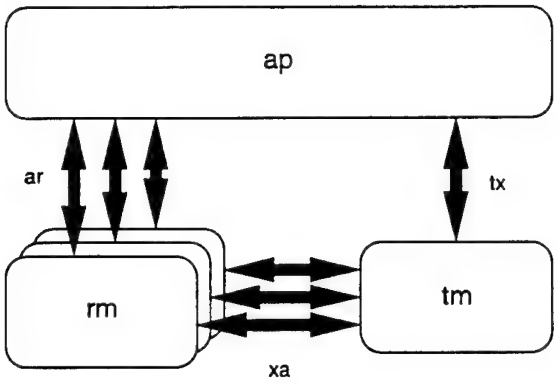

Figure 6.1: Abstract SDTP Architecture — components linked by secure channels

properties, to generate a purported formal proof that the compiled code has those properties as part of the code generation process. The purported proof can then be checked and, if it turns out to be a correct proof, it follows that the generated code has the desired properties. Thus, the emphasis is shifted from showing that compiler transformations are correct in general to checking that they produced correct results in individual cases.

The application of this idea to architectural transformation is straightforward. At some abstract level, the architectural description is proven to guarantee that the architecture has some desirable property, π. The interpretation mapping $\mathscr{I}$ that sends abstract-level sentences to their implementations can also be applied to the proof of π. If the image of the proof under the implementation mapping turns out to be a correct proof that the implementation has $\mathscr{I}(\pi)$, then, of course, the implementation has $\mathscr{I}(\pi)$. Checking the transformed proof can, therefore, provide the desired correctness guarantee.

6.3 An Example: secure distributed transaction processing

The idea of proof-carrying architectures can be illustrated by consideration of an example, based on our development of software architectures for secure distributed transaction processing (SDTP) [27]. These architectures extend X/Open's standard DTP architecture [46] by enforcing a simple "no read up, no write down" security policy. Figure 6.1 depicts an abstract architecture for SDTP. The boxes are the components of the architecture: the Application (labeled "ap"), some number of Resource Managers (labeled "rm"), and a Transaction Manager (labeled "tm"). The components are linked by *secure channels*, indicated by the heavy doubleheaded arrows that make up the interfaces between

{1}	1.	$(\forall d : \text{Labeled\_Data})[\text{Flows}(d, rm, ap) \rightarrow \text{Carries}(\text{secure\_ar\_channel}, d, rm\text{'s\_ar\_port}, ap\text{'s\_ar\_ports}(rm))]$	Axiom describing specific architecture
{2}	2.	$\text{Port\_Of}(ap\text{'s\_ar\_ports}(rm), ap)$	Axiom describing specific architecture
{3}	3.	$(\forall c : \text{Secure\_Channel})(\forall d : \text{Labeled\_Data})(\forall x : \text{Output\_Port})(\forall y : \text{Input\_Port})[\text{Carries}(c, d, x, y) \rightarrow \text{label}(d) \leq \text{clearance}(y)]$	Axiom characterizing secure channels
{4}	4.	$(\forall a : \text{Component})(\forall y : \text{Input\_Port})[\text{Port\_Of}(y, a) \rightarrow \text{clearance}(y) \leq \text{clearance}(a)]$	Axiom constraining port clearances
{5}	5.	$(\forall s_1, s_2, s_3 : \text{Security\_Label})[s_1 \leq s_2 \wedge s_2 \leq s_3 \rightarrow s_1 \leq s_3]$	Axiom specifying transitivity of security label ordering
{1}	6.	$\text{Flows}(d_0, rm, ap) \rightarrow \text{Carries}(\text{secure\_ar\_channel}, d_0, rm\text{'s\_ar\_port}, ap\text{'s\_ar\_ports}(rm))$	Universal instantiation (1)
{3}	7.	$\text{Carries}(\text{secure\_ar\_channel}, d_0, rm\text{'s\_ar\_port}, ap\text{'s\_ar\_ports}(rm)) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm))$	Universal instantiation (3)
{1,3}	8.	$\text{Flows}(d_0, rm, ap) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm))$	Tautological consequence (6,7)
{4}	9.	$\text{Port\_Of}(ap\text{'s\_ar\_ports}(rm), ap) \rightarrow \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \leq \text{clearance}(ap)$	Universal instantiation (4)
{2,4}	10.	$\text{clearance}(ap\text{'s\_ar\_ports}(rm)) \leq \text{clearance}(ap)$	Tautological consequence (2,4)
{5}	11.	$\text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \wedge \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \leq \text{clearance}(ap) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap)$	Universal instantiation (5)
{2,4,5}	12.	$\text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap)$	Tautological consequence (10,11)
{1,2,3,4,5}	13.	$\text{Flows}(d_0, rm, ap) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap)$	Tautological consequence (8,12)
{1,2,3,4,5}	14.	$(\forall d : \text{Labeled\_Data})[\text{Flows}(d, rm, ap) \rightarrow \text{label}(d) \leq \text{clearance}(ap)]$	Universal generalization (13)

Figure 6.2: Formal Proof that Dataflow from rm to ap Satisfies the Security Policy

the Application and Resource Managers, the Application and the Transaction Manager, and the Resource Managers and the Transaction Manager. A secure channel is a connector that enforces the security policy. In other words, secure channels will not carry classified data from a component to a component that lacks required clearances. To say that the system as a whole satisfies the security policy means that there is no flow of classified data to a component that lacks the required clearances. The security of the system follows almost immediately from the fact that it employs only secure channels, making a textbook-style natural deduction proof [18, 20] of system security quite simple. Consider dataflow from some Resource Manager rm to the Application ap, for example. A proof of the formula

$$(\forall d : \text{Labeled\_Data})[\text{Flows}(d, rm, ap) \rightarrow \text{label}(d) \leq \text{clearance}(ap)]$$

which says

> every labeled datum d that flows from rm to ap has a security label classifying it that is less than or equal to the clearance level of ap

from five axioms of the architectural theory is shown in Figure 6.2.

The five axioms say

1. every labeled datum d that flows from rm to ap is carried by secure_ar_channel from the output port rm's_ar_port to the input port of the port array ap's_ar_ports that is indexed by rm,

2. the input port of the port array ap's_ar_ports that is indexed by rm is a port of ap,

3. if secure channel c carries labeled datum d from output port x to input port y, then d's security label is less than or equal to the clearance level of y,

4. the clearance level of any port y of component a must be less than or equal to the clearance level of a, and

5. the ordering of security labels is transitive.

The first two axioms are facts about the particular architecture, the third axiom is the defining property of the secure channel subtype, the fourth and fifth axioms are general axioms of the security model.

The secure channels of abstract SDTP architecture can be implemented in terms of ordinary dataflow channels and additional components in a variety of ways, depending upon the security properties of the components [27]. The most interesting implementation is shown in Figure 6.3. This implementation is suited to the case where all of the resource managers and the application are single-level, but not necessarily the same level. The security policy is enforced by a multilevel secure component that filters dataflow between the application and the resource managers: if passing a datum from a resource manager to the application would violate the security policy, the filter removes it from the stream.

A proof that this architecture implements the more abstract architecture is not difficult (cf. [27]). Intuitively, it amounts to showing that the combination of a channel, a multilevel secure (MLS) filter, and a channel is a correct implementation of a secure channel. Ideally, the system architect should not be burdened with attempting to produce such proofs. The usual way of avoiding the necessity of proving correctness in this particular case is to prove that the Filter Introduction Transformation (FIT) — which can be informally represented as

$$\langle \textit{secure channel} \rangle \longrightarrow \langle \textit{channel} \rangle + \langle \textit{filter} \rangle + \langle \textit{channel} \rangle$$

where + indicates connection of a connector to a component — always produces correct implementations. But, if FIT has not been generally verified, how can the architect who is convinced of its correctness, at least in the particular case, proceed?

If the architect is using a transformation system that produces "proof-carrying architectures", when transformation FIT is applied, it is applied not only to the architectural description, but to the formal security proof of Figure 6.2 as well. The result of applying FIT to this proof is shown in Figure 6.4,

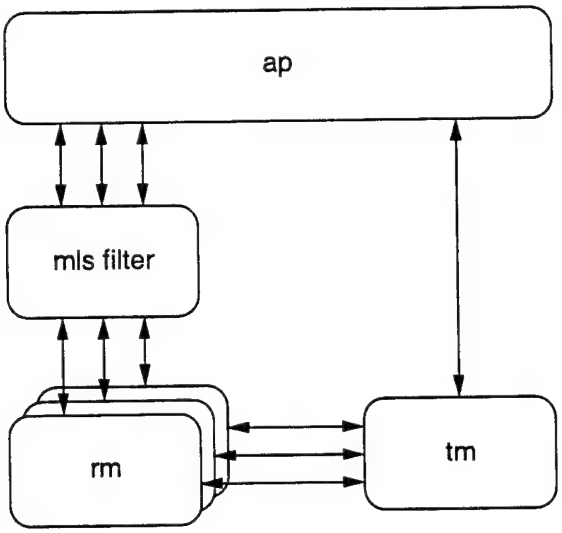

Figure 6.3: More Concrete SDTP Architecture — secure channels refined to ordinary channels, or ordinary channels plus security filter

where the implementation mapping $\mathscr{I}$ associated with this application is determined as follows. The Carries predicate

$$\text{Carries}(\langle \textit{secure channel} \rangle, \langle \textit{datum} \rangle, \langle \textit{out port} \rangle, \langle \textit{in port} \rangle)$$

that is mentioned in formulas 1, 3, 6, and 7 of the abstract-level proof is mapped to a conjunction of the Carries, Passes, and Carries predicates,

$$\begin{aligned} &\text{Carries}(\langle \textit{channel} \rangle, \langle \textit{datum} \rangle, \langle \textit{out port} \rangle, \langle \textit{filter in port} \rangle) \\ &\quad \wedge \text{Passes}(\langle \textit{filter} \rangle, \langle \textit{datum} \rangle, \langle \textit{filter in port} \rangle, \langle \textit{filter out port} \rangle) \\ &\quad \wedge \text{Carries}(\langle \textit{channel} \rangle, \langle \textit{datum} \rangle, \langle \textit{filter out port} \rangle, \langle \textit{in port} \rangle) \end{aligned}$$

This clause in the definition of $\mathscr{I}$ says that a secure channel carrying a datum from some output port to some input port is implemented as a channel carrying the datum from the output port to some input port of a filter, passing the datum through the filter from the input port to some output port, and carrying the datum from the output port of the filter to the input port. This mapping would not be appropriate to apply to every occurrence of the Carries predicate in every derivation, because not every secure channel in the abstract architecture is being replaced by a combination of two channels and a filter in the concrete architecture. However, formulas 1, 6, and 7 of the proof specifically refer to what secure_ar_channel carries, and this secure channel *is* being implemented by two channels and a filter. This mapping is also applied to formula 3 in order

{1}	1.	$(\forall d : \text{Labeled\_Data})[\text{Flows}(d, rm, ap)$
		$\rightarrow \text{Carries}(rm\_to\_filter\_channel, d, rm\text{'s\_ar\_port}, filter\_in\_port(rm))]$
		$\wedge \text{Passes}(mls\_filter, d, filter\_in\_port(rm), filter\_out\_port(rm))]$
		$\wedge \text{Carries}(filter\_to\_ap\_channel(rm), d, filter\_out\_port(rm), ap\text{'s\_ar\_ports}(rm))]$
{2}	2.	$\text{Port\_Of}(ap\text{'s\_ar\_ports}(rm), ap)$
{3}	3.	$(\forall c_1 : \text{Channel})(\forall f : \text{MLS\_Component})(\forall c_2 : \text{Channel})(\forall d : \text{Labeled\_Data})$
		$(\forall x_1 : \text{Output\_Port})(\forall x_2 : \text{Output\_Port})(\forall y_1 : \text{Input\_Port})(\forall y_2 : \text{Input\_Port})$
		$[\text{Carries}(c_1, d, x_1, y_1) \wedge \text{Passes}(f, d, y_1, x_2) \wedge \text{Carries}(c_2, d, x_2, y_2)$
		$\rightarrow \text{label}(d) \leq \text{clearance}(y_2)]$
{4}	4.	$(\forall a : \text{Component})(\forall y : \text{Input\_Port})[\text{Port\_Of}(y, a) \rightarrow \text{clearance}(y) \leq \text{clearance}(a)]$
{5}	5.	$(\forall s_1, s_2, s_3 : \text{Security\_Label})[s_1 \leq s_2 \wedge s_2 \leq s_3 \rightarrow s_1 \leq s_3]$
{1}	6.	$\text{Flows}(d_0, rm, ap)$
		$\rightarrow \text{Carries}(rm\_to\_filter\_channel, d_0, rm\text{'s\_ar\_port}, filter\_in\_port(rm))$
		$\wedge \text{Passes}(mls\_filter, d_0, filter\_in\_port(rm), filter\_out\_port(rm))$
		$\wedge \text{Carries}(filter\_to\_ap\_channel(rm), d_0, filter\_out\_port(rm), ap\text{'s\_ar\_ports}(rm))$
{3}	7.	$\text{Carries}(rm\_to\_filter\_channel, d, rm\text{'s\_ar\_port}, filter\_in\_port(rm)$
		$\wedge \text{Passes}(mls\_filter, d_0, filter\_in\_port(rm), filter\_out\_port(rm))$
		$\wedge \text{Carries}(filter\_to\_ap\_channel(rm), d_0, filter\_out\_port(rm), ap\text{'s\_ar\_ports}(rm))$
		$\rightarrow \text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm))$
{1,3}	8.	$\text{Flows}(d_0, rm, ap) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm))$
{4}	9.	$\text{Port\_Of}(ap\text{'s\_ar\_ports}(rm), ap) \rightarrow \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \leq \text{clearance}(ap)$
{2,4}	10.	$\text{clearance}(ap\text{'s\_ar\_ports}(rm)) \leq \text{clearance}(ap)$
{5}	11.	$\text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \wedge \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \leq \text{clearance}(ap)$
		$\rightarrow \text{label}(d_0) \leq \text{clearance}(ap)$
{2,4,5}	12.	$\text{label}(d_0) \leq \text{clearance}(ap\text{'s\_ar\_ports}(rm)) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap)$
{1,2,3,4,5}	13.	$\text{Flows}(d_0, rm, ap) \rightarrow \text{label}(d_0) \leq \text{clearance}(ap)$
{1,2,3,4,5}	14.	$(\forall d : \text{Labeled\_Data})[\text{Flows}(d, rm, ap) \rightarrow \text{label}(d) \leq \text{clearance}(ap)]$

Figure 6.4: Transformed Formal Proof that Dataflow from rm to ap Satisfies the Security Policy

to preserve the fact that formula 7 should follow from formula 3 by Universal Instantiation. The universal quantifier over secure channels in formula 3,

$$(\forall \langle \textit{secure channel variable} \rangle : \text{Secure\_Channel})$$

is mapped by $\mathscr{I}$ to universal quantifiers over channels and a universal quantifier over MLS components,

$$\begin{array}{l}(\forall \langle \textit{to-filter channel variable} \rangle : \text{Channel}) \\ \quad (\forall \langle \textit{filter variable} \rangle : \text{MLS\_Component}) \\ \quad (\forall \langle \textit{from-filter channel variable} \rangle : \text{Channel})\end{array}$$

It is easy to check that the result of applying the FIT interpretation mapping $\mathscr{I}$ to the proof of security is a syntactically correct derivation of the desired security property from formulas that are images of axioms of the more abstract architectural theory. Mapping $\mathscr{I}$ sends tautological consequence steps to correct tautological consequence steps, universal instantiation steps to correct universal instantiation steps, and universal generalization steps to correct universal generalization steps. So $\mathscr{I}$ has indeed mapped the formal abstract-level security proof to a concrete-level security proof, but not necessarily a proof *from axioms of the concrete architectural theory.*

$\{1\}$ 1. $(\forall c : \mathsf{Channel})(\forall d : \mathsf{Labeled\_Data})(\forall x : \mathsf{Output\_Port})(\forall y : \mathsf{Input\_Port})$
$[\mathsf{Carries}(c, d, x, y) \rightarrow \mathsf{clearance}(x) = \mathsf{clearance}(y)]$
Axiom specifying connection constraint imposed by security model

$\{2\}$ 2. $(\forall f : \mathsf{MLS\_Component})(\forall d : \mathsf{Labeled\_Data})(\forall y : \mathsf{Input\_Port})(\forall x : \mathsf{Output\_Port})$
$[\mathsf{Passes}(f, d, y, x) \rightarrow \mathsf{label}(d) \leq \mathsf{clearance}(x)]$
Axiom characterizing MLS components

$\{3\}$ 3. $(\forall x)(\forall y)(\forall z)[x = y \rightarrow [z \leq x \leftrightarrow z \leq y]]$ Instance of identity axiom schema

$\{1\}$ 4. $\mathsf{Carries}(c_2, d_0, x_2, y_2) \rightarrow \mathsf{clearance}(x_2) = \mathsf{clearance}(y_2)$ Universal instantiation (1)

$\{2\}$ 5. $\mathsf{Passes}(f_0, d_0, y_1, x_2) \rightarrow \mathsf{label}(d_0) \leq \mathsf{clearance}(x_2)$ Universal instantiation (2)

$\{1,2\}$ 6. $\mathsf{Carries}(c_1, d_0, x_1, y_1) \wedge \mathsf{Passes}(f_0, d_0, y_1, x_2) \wedge \mathsf{Carries}(c_2, d_0, x_2, y_2)$
$\rightarrow \mathsf{label}(d_0) \leq \mathsf{clearance}(x_2) \wedge \mathsf{clearance}(x_2) = \mathsf{clearance}(y_2)$
Tautological consequence (3,4)

$\{3\}$ 7. $\mathsf{clearance}(x_2) = \mathsf{clearance}(y_2) \rightarrow [\mathsf{label}(d_0) \leq \mathsf{clearance}(x_2) \leftrightarrow \mathsf{label}(d_0) \leq \mathsf{clearance}(y_2)$
Universal instantiation (3)

$\{1,2,3\}$ 8. $\mathsf{Carries}(c_1, d_0, x_1, y_1) \wedge \mathsf{Passes}(f_0, d_0, y_1, x_2) \wedge \mathsf{Carries}(c_2, d_0, x_2, y_2)$
$\rightarrow \mathsf{label}(d_0) \leq \mathsf{clearance}(y_2)$
Tautological consequence (6,7)

$\{1,2,3\}$ 9. $(\forall c_1 : \mathsf{Channel})(\forall f : \mathsf{MLS\_Component})(\forall c_2 : \mathsf{Channel})(\forall d : \mathsf{Labeled\_Data})$
$(\forall x_1 : \mathsf{Output\_Port})(\forall x_2 : \mathsf{Output\_Port})(\forall y_1 : \mathsf{Input\_Port})(\forall y_2 : \mathsf{Input\_Port})$
$[\mathsf{Carries}(c_1, d, x_1, y_1) \wedge \mathsf{Passes}(f, d, y_1, x_2) \wedge \mathsf{Carries}(c_2, d, x_2, y_2)$
$\rightarrow \mathsf{label}(d) \leq \mathsf{clearance}(y_2)]$
Universal generalization (8)

Figure 6.5: Proof of Image of Abstract-Level Formula 3 under $\mathscr{I}$ from Axioms of Concrete Theory

The image of the first axiom under $\mathscr{I}$ says that every labeled datum that flows from rm to ap is carried to the filter from rm, passed through the filter, and then carried to ap from the filter. Just as in the case of the first axiom, this is a fact about the particular architecture that is either an axiom of the concrete theory, or easily and automatically derivable from axioms of the concrete theory. The mapping $\mathscr{I}$ leaves the second axiom unchanged. This will certainly be an axiom of the concrete theory, as well as the abstract theory. The image of the third axiom is a bit more complex. It states that the combination of the two channels and the filter enforces the security property. It is quite unlikely that this would be among the chosen axioms of the concrete-level theory, since it is the filter alone, effectively, that is enforcing security. Still, it is easy to see that this formula must be a consequence of axioms of the concrete theory: the security model requires that channels which do not enforce security can only connect ports with matching clearances, and one of the defining properties of an MLS component is that it supplies data at an output port only if the classification of the data is less than or equal to the clearance of the port. A formalization of this proof is shown in Figure 6.5. Discovery of this proof is easy. The form of the desired conclusion — a conjunction of conditions on Carries and Passes in the antecedent, and the comparison of label to clearance in the consequent — immediately suggests the use of the axioms on lines 1 and 2 of the proof. So it should be quite plausible that the proof can be discovered without human intervention by the transformation system. The interpretation mapping $\mathscr{I}$ does not affect the images of the remaining two axioms; they remain general axioms of the security model. So, by combining the proof in Figure 6.5 with the proof in Figure 6.4, we obtain a proof of the security property from axioms

of the concrete theory. Moreover, this proof is recognizably a formalization of our informal argument [27, p. 89] that the concrete architecture satisfies the security policy.

6.4 Generalizing from the Example

The idea of using the architectural transformation to transform the proof that the more abstract architecture has a desired property into a proof that the more concrete architecture has the property worked well for this rather simple example. Is there any reason to believe that it will work equally well in other cases?

Recall that the standard criterion for correctness of an implementation mapping $\mathscr{I}$ of an abstract logical theory Θ (of first-order logic) in a more concrete (first-order) theory Θ' is that $\mathscr{I}$ must interpret Θ in Θ', i.e., it must be the case that, for every formula φ in the language of Θ,

$$\Theta \vDash \varphi \quad \Longrightarrow \quad \Theta' \vDash \mathscr{I}(\varphi)$$

The basic facts about theory interpretation can be found in the logic textbooks of Enderton [8] and Shoenfield [44], among others. For present purposes, it is enough to know that

1. for every formula φ of the language of the more abstract theory,

   $$\mathscr{I}(\neg\varphi) = \neg\mathscr{I}(\varphi)$$

   and similarly for the other connectors, and

2. for every formula φ of the language of the more abstract theory, and every variable $\mathbf{x}$,

   $$\mathscr{I}(\forall \mathbf{x}\, \varphi) = \forall \mathbf{x}\, [\omega_{\mathscr{I}}(\mathbf{x}) \rightarrow \mathscr{I}(\varphi)]$$

   where $\omega_{\mathscr{I}}(\mathbf{x})$ is some formula determined by $\mathscr{I}$, not dependent upon φ, in which only $\mathbf{x}$ occurs freely, and similarly for the other quantifiers.

If $\mathscr{I}$ interprets Θ in Θ', an easy inductive argument shows that $\mathscr{I}$ maps formal proofs from Θ to formal proofs from $\mathscr{I}[\Theta]$ that can be extended to proofs from Θ'. If φ is an axiom of Θ, then, since $\mathscr{I}$ is a theory interpretation, $\mathscr{I}(\varphi)$ is derivable from Θ'. Because $\mathscr{I}$ is defined so that connectives pass through it, $\mathscr{I}$ maps tautological consequence steps to tautological consequence steps. Because $\mathscr{I}$ uniformly restricts quantifiers, $\mathscr{I}$ maps universal instantiation and universal generalization steps to universal instantiation and generalization steps, respectively. Thus, $\mathscr{I}$ maps formal proofs from abstract axioms to formal proofs from images of abstract axioms, and images of abstract axioms can always be proved from concrete axioms.

So, if an architectural transformation step is correct, in the standard sense, the corresponding interpretation mapping will map formal proofs to formal

proofs containing small gaps. A fortiori, an abstract-level formal proof of some particular property of interest — say, satisfaction of a security policy — will be mapped to a proof that the implementation also has (the implementation-level analogue of) the property. Since the replacement of the secure channel from rm to ap by a pair of channels and a filter is evidently correct, it is not surprising that the FIT mapping sends the abstract-level security proof to a concrete-level security proof.

It follows that the proof-carrying architecture approach allows the architect to perform arbitrary correct transformations when implementing an abstract architecture, provided the transformation system that supports the approach is clever enough to find the proofs of images of axioms. Of course, incorrect transformations that happen to preserve the proof of the property of interest are also allowed. Therefore, this approach is well-suited to the case where the focus is on obtaining an implementation with some particular desirable property — i.e., when a weaker-than-usual correctness criterion is adequate — and placing minimal constraints on the architect's implementation options is preferred.

6.5 Related Work

Although there is a large and growing literature on formal software transformation, nearly all of it is oriented toward maintaining functional correctness, rather than system structure. Similarly, there is a large body of literature on architectural refinement and composition, nearly all of it employing semiformal representation and analysis techniques, at best. The comparatively few papers on formal refinement of architectural structure include Broy's work on component refinement [4], Brinksma, et al.'s, work on connector refinement [3], Philipps and Rumpe's recent work on refinement of information flow architectures [35], and the work described in our own earlier papers [25, 26, 27, 39, 41]. Also closely related is work by Garlan's group [1], Luckham's group [19], and Moriconi and Qian [25] on formally representing the semantics of connectors and relating semantic models at different levels of abstraction. But, the emphasis in all these cases has always been on verification of general refinement patterns, rather than checking particular steps.

Necula and Lee's work on proof-carrying code and its applications [31, 32, 30] introduced the notion of replacing verification by checking in the context of compilation. The work described in this paper can be viewed as generalizing their ideas about code refinement transformations to architectural transformations, both refinements and abstractions.

6.6 Chapter Summary

Transformational development of architectures can guarantee that implementations are correct by restricting the architect to a stock of verified transformations. But such a correctness guarantee is quite brittle, since use of a single

non-verified transformation voids the guarantee. Moreover, confidence that the implementation is correct should be no greater than confidence that all the verifications are correct. If many transformations are used, and the verification of each is difficult, then confidence in the correctness of the implementation may be less than desired, especially in safety-critical applications. *Checking* particular refinement steps offers a way of allowing the architect greater freedom, and of achieving higher levels of confidence that the implemented architecture has the desired properties.

Our initial approach to checking, based on the idea of proof-carrying architectures, is especially well suited to the case where the main requirement is high confidence that the implementation has some specific property. The property is shown to hold at some abstract level, and every refinement is produced by application of a transformation known to preserve the property, or is checked for correctness by making sure that the transformation preserves the proof of the desired property, or both.

The main limitation of this first approach to checking is that only a limited range of properties can be checked. The approach is too weak to ensure that an implementation step is correct, in the standard sense, which requires preservation of a broad range of properties. For that reason, we are exploring other approaches to checking in parallel with the proof-carrying architectures research. An alternative approach that seems particularly promising is based on the idea of applying the simplified technique for proving implementation mapping correctness [39] at architecture-definition-time to particular development steps. This approach will, in certain circumstances, allow checking that particular steps are correct, according to our strong correctness criterion, rather than checking one or a few properties of interest.

Proof-carrying architectures are being implemented as part of the Xform[3] system. *Xform*, pronounced *transform*, is a recursive acronym for "*Xform, for orderly reification*[4] *and maintenance.*" Xform is a toolset for transformational development and maintenance of architectural descriptions written in languages such as SADL [28] and ACME [11].

[3] *Xform* is a common mathematical shorthand for *transform*.

[4] *To reify* means to make actual. Thus, reification of software architectures is the process of turning them into actual implementations.

Chapter 7

Overview of the SDTP Derivation

7.1 Introduction

The primary focus of our research efforts is on the problem of producing *dependable system architectures.* We have recently developed the basic technology required to effectively solve this problem in the case of distributed transaction processing systems. Our solution involves extending the ACID properties that drove the development of the X/Open standard for distributed transaction processing by adding security constraints. Specifically, we added the constraint that the system must satisfy a multilevel secure access control policy. Our basic approach to defining a family of architectures that satisfy the extended requirements was described in an earlier paper [27]. This chapter is devoted to describing our experiences in following that approach to its logical conclusion: an implemented architecture for secure distributed transaction processing that has been proven secure.

7.2 Architecture Hierarchies

It is common practice to informally describe a single architecture at many levels of abstraction, and from a variety of different perspectives. For example, at a concrete level, the architecture might be described in terms of how a system is assembled from code components using connection mechanisms provided by a programming language or operating system. At a very abstract level, the same architecture might be described in terms of the data and control flow paths that link the functions the system is required to perform. At an intermediate level, the system architecture might be described in terms of clients and servers interacting through message queues.

One reason that multiple descriptions are provided is that each of these archi-

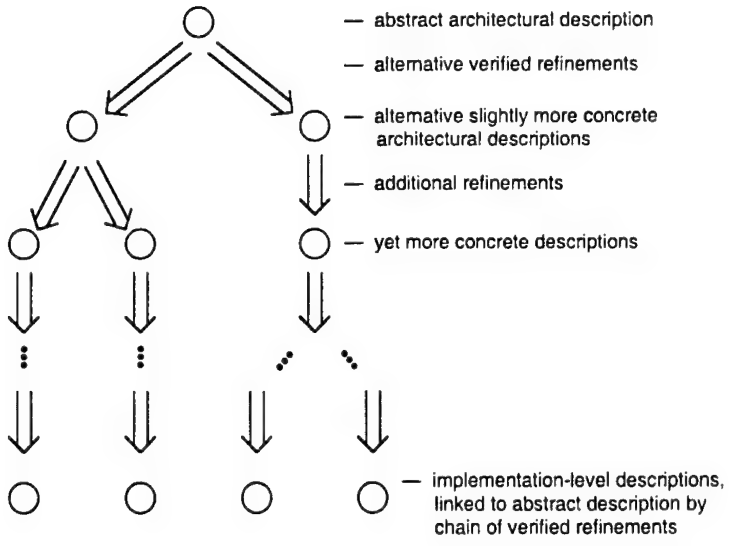

Figure 7.1: An Architecture Hierarchy

tectural descriptions is useful for some purposes, but less useful for others. Often the argument that the system has some desirable architectural property can be greatly simplified by employing an abstract description of the architecture. On the other hand, some properties require detailed analysis of low-level architectural descriptions. Another reason is that there is often a major conceptual gap between abstract architectural descriptions and the actual implementation. Descriptions at intermediate levels of abstraction break this gap down into intellectually tractable pieces, and thus can significantly increase confidence that the abstract description correctly characterizes the as-implemented architecture.

While much of the recent work on more formal architectural description languages (ADLs) has focused on the problem of describing architectures in a way that provides a suitable basis for various formal analyses, our own work has focused primarily on the problem of formally linking these several descriptions in an *architecture hierarchy*. The basic idea is to regard each description as a logical theory, the theory of the class of architectural structures that the description describes, and to formalize the links between the descriptions as interpretation mappings. Figure 7.1 shows the structure of a typical architecture hierarchy. Correctness of the hierarchy can be explicated in terms of constraints on the interpretation mappings. For example, one might require that the interpretations preserve some property of special interest (say, some security property). Or one might require that they preserve every property expressible in some language.

Our tools for constructing architecture hierarchies are based upon an incremental transformation paradigm. The result of applying a transformation to a hierarchy is a hierarchy containing additional or altered descriptions. An additional description is typically either an abstraction of another description

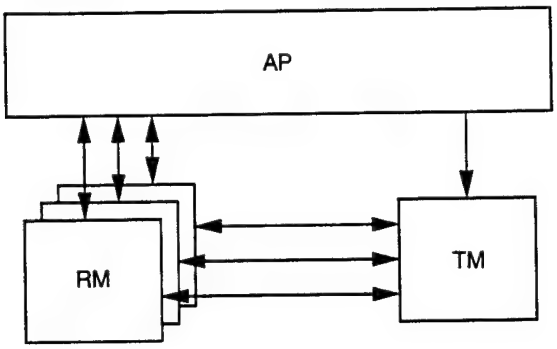

Figure 7.2: The X/Open DTP Reference Architecture

in the hierarchy, a refinement of another description in the hierarchy, or a description at the same level of abstraction but from a different perspective (e.g., a data-oriented description that complements an existing function-oriented description). If a hierarchy has been constructed entirely by applications of transformations that are known to be generally correctness-preserving, or whose correctness in the particular instance is checked as they are applied, the hierarchy is known to be correct. Correct hierarchies formally link very abstract formal architectural descriptions, which can easily be demonstrated to satisfy system requirements, to implementation-level architectural descriptions in a way guaranteeing that the implementation satisfies the requirements.

7.3 Distributed Transaction Processing

In response to a growing demand for increased interoperability of software components, several vendor-neutral, open software architectures have been proposed as standards. The X/Open Distributed Transaction Processing (DTP) reference architecture [46] is one of these standards. X/Open DTP is specifically intended to standardize interactions among the components of a three-tiered client/server model: resource (e.g., database) managers (RMs), an application (AP) that accesses those resources, and a transaction manager (TM). A very abstract, informal representation of this architecture — which can be found in the X/Open documents — is shown in Figure 7.2.

The role of the TM is to ensure that consistency is maintained when the state of the RMs is changed by the AP. Transactions are collections of operations on the RMs. Multiple transactions can be performed concurrently. In the context of a transaction, all other transactions appear to be executed atomically. In other words, it always appears that all or none of the operations that make up a transaction have taken effect.

The X/Open documentation is informal, consisting of a mix of C header

files[1] and English text describing the semantics of the functions declared in the headers. The protocols for using these functions are also only semiformally characterized.

7.4 Adding Security to DTP

Security requirements are ubiquitous in transaction processing, both within the defense sector and within the commercial sector. Tremendous leverage can be gained by addressing security concerns at the architectural level, because ensuring security is typically both technically challenging and expensive. Several of the open software architectures have introduced extensions to deal with security issues. For example, recent updates of OMG's CORBA specification have included security services that address fundamental security concerns in a distributed object system, based on a trusted ORB model. These services include credential-based authentication of principals and their clients, several simple privilege delegation schemes, authorization based on access control lists, audit trails, and non-repudiation services based on the ISO non-repudiation model.

One of the key security issues in DTP is enforcement of an *access control policy*, which ensures that classified resoures can be accessed only by clients with appropriate clearances. In terms of the Bell-LaPadula model [17], our proposed multilevel secure (MLS) access control policy for secure DTP can be defined by saying that the distributed transaction processing systems must have the following two properties:

- *The Simple Security Property* — A client is allowed read access to a resource only if the client's clearance level is greater than or equal to the resource's classification level.
- *The $\star$-Property* — A client is allowed write access to a resource only if the client's clearance level is less than or equal to the resource's classification level.

The simple security property guarantees that there is no *read-up* of data, while the $\star$-property guarantees that there is no *write-down*.

To say that a component in a DTP system is MLS means that the component satisfies the MLS access policy internally, and that all inputs to and outputs from the component are properly classified. So, for example, to say that a multilevel database management system (DBMS) is an MLS resource manager means that

- any data entered into the database is tagged with a classification level,
- any data retrieved from the database comes tagged with the same classification level that it had when it was entered, and

[1] Since a description of how components written in languages other than C, including languages such as COBOL that do not support return values, can be accommodated is included, it is clear that the C code cannot be considered to be a formal specification of the syntax of the interface functions.

- no operation performed by the DBMS can result in either read-up or write-down among the subcomponents of the DBMS.

(Note that a single-level database is trivially an MLS resource manager, provided single-levelness is maintained.) Similarly, to say that the entire DTP system is MLS means that no read-up or write-down occurs between components and that inputs and outputs are properly classified.

The challenge in adding an MLS access control policy to X/Open DTP is to define an architecture which guarantees that, if the components used to build the system are MLS, then the DTP system as a whole is MLS. That this is not a trivial problem can be seen by considering a DTP system composed of an application that accesses several single-level databases, each of which has a different classification level, a very common situation. A secure DTP (SDTP) architecture must ensure that the client application does not read data classified above the application's clearance level, that it does not write data that it obtained from a highly classified database to a database classified at a lower level, and so on.

One complicating factor is that no single architectural structure is well suited to every SDTP system. It is easy to see that a single-level client AP, a single-level TM, and a collection of single-level RMs all at the same level can be linked by secure connectors to build a system that is automatically MLS secure. Extending the X/Open DTP architecture to handle such systems can be accomplished simply by adding appropriate encryption to the communication protocols. But varying single-level systems will require that the connectors linking the AP to the RMs somehow enforce the access control policy (at least in the absence of a certification that the AP satisfies security constraints beyond being MLS). One way of doing so is to introduce a Security Manager component that mediates communication between the RMs and the MLS AP, as shown in Figure 7.3.

7.5 Defining the SDTP Hierarchy

As part of an earlier effort to develop an architectural description language suitable for defining architectural hierarchies, we defined a formal hierarchy for the X/Open DTP standard consisting of 17 SADL [28][2] specifications, linked by SADL-specified interpretation mappings. The most abstract description, shown in Figure 7.4, approximately corresponds to the informal diagram shown in Figure 7.2. Six levels below the most abstract description is a description at roughly the same level of abstraction as the code in the X/Open documentation. (Excerpts from this code are shown in Figure 7.5.) Below this level, the hierarchy branches in several directions, reflecting, e.g., the various choices that can be made in allocating low-level interprocessor communication functions to processors.

[2]SADL, pronounced *saddle*, is an acronym for 'Structural Architecture Description Language'. The name was suggested by the fact that SADL emphasizes description of structure rather than description of the behavior of connectors, a topic quite adequately addressed by other ADLs.

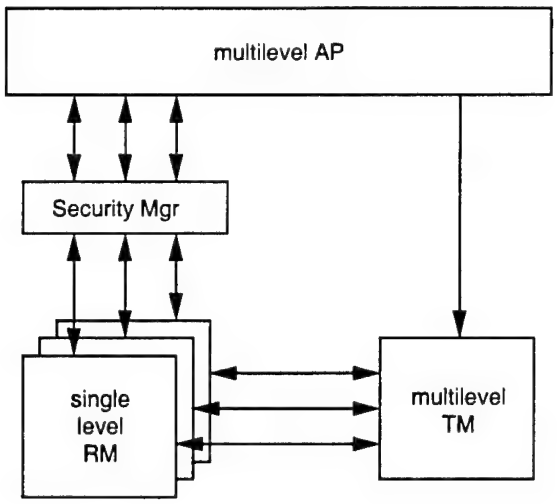

Figure 7.3: SDTP Architecture for Varying Single-level Resource Managers

```
x_open_abstract:  ARCHITECTURE [ -> ]
  IMPORTING ALL FROM Dataflow_Relations_style
BEGIN

CONFIGURATION
  ap:  TYPE <= Function
  rms:  TYPE <= Function
  tm:  TYPE <= Function

  the_ap:  ap
  the_rms:  rms
  the_tm:  tm

  ar:  CONSTRAINT = Dataflow(the_ap, the_rms)
  tx:  CONSTRAINT = Dataflow(the_ap, the_tm)
  xa:  CONSTRAINT = Dataflow(the_tm, the_rms)

END x_open_abstract
```

Figure 7.4: Dataflow-Level SADL Description of X/Open Architecture

```
x_open_std_level:  ARCHITECTURE [ -> ]
%% X_Open_style defines relevant
%%  properties of procedure calls;
%%  this description covers how things
%%  are hooked together
  IMPORTING ALL FROM X_Open_style , ...
BEGIN
  ...
COMPONENTS
  tm:  TYPE <= ARCHITECTURE [ -> ]
     EXPORTING ALL
     BEGIN
       register:  AX_Register_Procedure
         [id:  X_Id, rmid:  INT, flags:  INT
                -> ret:  INT]
       begin:  TX_Begin_Proc [ -> ret:  INT]
       ...
     END tm
  ...
  the_tm:  tm
  ...

CONFIGURATION
  tx:  CONSTRAINT =
         Called_From(the_tm.begin, the_ap)
           AND ...
  xa:  CONSTRAINT =
        (FORALL y:  rm)
          [Called_From(the_tm.register, y)
            AND ...
  ...
END x_open_std_level
```

Figure 7.5: X/Open Standard-Level SADL Description of X/Open Architecture (Heavily Elided)

Our basic approach to developing a similar hierarchy for SDTP architectures was to treat SDTP as a *rearchitecting* problem: given an architecture that satisfies a set of requirements, how can that architecture be modified to satisfy an additional requirement, viz., that the MLS access policy is satisfied. The idea is to attempt to "replay" the derivation of the implementation-level architecture descriptions from the most abstract architectural description. An architecture hierarchy includes a record of the transformations used to generate the interpretations that link the descriptions. If a transformation used in the original X/Open DTP hierarchy can be shown to preserve satisfaction of the additional requirement — i.e., in this case, if the transformation can be shown to preserve the simple security property (no read-up) and the $\star$-property (no write-down) — then the same transformation can be employed in deriving an SDTP implementation.[3]

A priori, it was not known what percentage of the design history encoded in the X/Open DTP hierarchy could be reused in the SDTP hierarchy. We therefore developed implementation-level descriptions of the three abstract architectures for SDTP that we have described in an earlier publication [27]. It turned out that over 90% of the design decisions made during the implementation of each of the three SDTP architectures could be based on decisions recorded in the DTP hierarchy; conversely, every design decision in the DTP hierarchy was reused in the development of every one of the three SDTP hierarchies. In the most extreme case, the hierarchy for the "single-level, same-level RMs" case is nearly identical to the DTP hierarchy, the only real difference being an additional low-level decision to use an RPC mechanism that performs encryption. Thus, one important result of this case study was evidence that confirms our hypothesis that derivation replay is often an effective rearchitecting technique.

7.6 Verifying Security

We believe that our implementation-level SDTP architectural descriptions define architectures that have the simple security property and the $\star$-property, because we have formally proved that these properties follow from a dataflow-level description of the SDTP architectures in terms of components linked by channels that enforce security and we are in the process of formally confirming our belief that the design decisions in the SDTP hierarchies preserve security. The proofs that the design decisions preserve the desired security properties use two different verification techniques we have developed.

Many of the transformations employed in development can be shown to preserve a broad class of structural properties by showing that the implementation links they introduce into the hierarchy are *faithful interpretations* of the theory of the more abstract description in the theory of the more concrete description.

[3]The bindings of the variables in the transformation can, and usual do, change. But the bindings in the replay can be automatically determined from the bindings in the original derivation by tracking the correspondence between objects in the two hierarchies as the new hierarchy is constructed from the old.

To say an interpretation $\mathscr{I}$ of the language of theory Θ_1 in theory Θ_2 is a faithful (theory) interpretation means that, for every sentence (i.e., closed formula) φ in the language of Θ_1,

$$\Theta_1 \vdash \varphi \quad \text{if and only if} \quad \Theta_2 \vdash \mathscr{I}(\varphi)$$

In general, access control policies prohibit certain dataflow paths in the system. Simply by omitting the undesired dataflow channels from the abstract architectural description, we guarantee that no formula saying *there is a dataflow from* C_1 *to* C_2, where a flow from component C_1 to component C_2 would violate the access control policy, can be proved from the theory of the abstract description. Faithful interpretations cannot introduce new dataflow channels, but can only further elaborate existing channels. Therefore, transformations that can be shown to always introduce faithful interpretations can freely be used in the process of implementing a design that has been verified to be secure, without fear that security will be violated. The details of our model-theoretic method for proving that transformations always introduce faithful interpretation links have been described in a series of earlier papers [26, 39, 41]. Currently, we manually prove that transformations introduce faithful theory interpretations, though we are actively investigating the possibility of using SRI's PVS verification system to provide automated support for the process.

On the other hand, not every interpretation link in the SDTP hierarchies has been shown to be a faithful theory interpretation. In fact, some of the transformations employed in building the hierarchies do not have the property that they always introduce faithful interpretations. For these interpretations, we apply a *checking* technique to show that the interpretation preserves the desired security properties. The basic idea is to use the interpretations that refine the architectural description to refine the proof that the descriptions have the desired security properties as well. If the result of applying the interpretation to the security proof at the more abstract level can be used to automatically obtain a security proof at the more concrete level, the implementation step has been shown to preserve security. (This technique has been described in greater detail in an earlier paper [40].) Currently, we apply the transformations to proofs manually, and then use PVS to check the results. A tool that automates proof transformation is under development.

7.7 Implementing an SDTP Reference Architecture

In this SDTP reference implementation the RMs consist of a Security Manager (SM) that enforces the desired multilevel security policies — each of which implements a piece of the Security Manager component shown in Figure 7.3 — and a set of single-level RMs, which are arbitrary single-level databases. A multilevel secure RM consists of the combination of the SM and the single-level RMs. Note that the single-level RMs are not required to provide the X/Open

DTP standard services themselves; they must, however, support transaction processing.

The X/Open DTP standard specifies various services that the components of the architecture must provide and make use of. An SDTP application will make use of two services: the **TX** (Transaction Demarcation) service, and the **AR** service (the service that allows the application access to shared resources). The **TX** service is provided by the TM while the **AR** service is provided by the (multilevel) RMs the application makes use of.

The TM and RMs also make use of X/Open DTP-specified services. The service the RMs provide to the TM is called the **xa** subservice of the **XA** service. This subservice implements the interface the TM uses to perform the two-phase commit protocol. The **ax** subservice of the **XA** service is provided by the TM and allows RMs to dynamically register themselves as being under the TM's management, and unregister themselves when that management is no longer desired.

The X/Open DTP standard gives a detailed specification of the **TX** and **XA** services. The **AR** service is considered implementation-dependent because RMs may provide services other than database management, and thus the **AR** interface must allow its clients to make use of those services.

Our reference implementation of SDTP was written in Common Lisp. The advantage of using a language quite different from X/Open's choice of C is that it helped us identify and eliminate some of the "C-centric" implementation decisions that crept into the X/Open design. Common Lisp was also attractive because it enabled rapid implementation and debugging of the system. The Common Lisp package system was used to globally identify the various services and prevent name conflicts (e.g., since **open** is a standard Common Lisp function for file I/O as well as appearing in both the **TX** and **XA-xa** service, a name conflict would occur unless some method of distinguishing between these functions was employed).

A remote procedure call (RPC) facility native to the Common Lisp version we used was employed for component-to-component communication. Communication between the RMs and the single-level databases was done by means of a foreign function interface (FFI) to the native database programmatic interface library.

7.7.1 The TX Service

The **TX** service allows the application to open and close resource managers, start, finish or roll back transactions, obtain information about the state of a transaction, and set certain transaction characteristics.

A typical chain of events involving the application making use of the **TX** service is

1. Tell the TM to open the RMs with the `tx:open` call.
2. Connect to the RMs and use the **AR** services of the RM for preliminary functions (such as authentication and key exchange).

3. Tell the TM to start a transaction with the `tx:begin` call.

4. Use the **AR** service of the RMs to exchange data.

5. Finish the transaction by using the TM's `tx:commit` call; alternatively, abort and roll back the transaction using the `tx:rollback` call.

6. Once the desired set of transactions is completed, tell the TM to close the RMs by using the `tx:close` call.

7.7.2 The XA Service

The **XA** service is used by the TM to communicate with the RMs and by the RMs to communicate with the TM. It is divided into two subservices as described above.

The **xa** subservice is used by the TM to communicate with the RMs to negotiate the two-phase commit protocol. A typical chain of events is

1. The application informs the TM that it wishes to begin processing transactions (via the **ax** subservice, described below).

2. The TM uses the `xa:open` call to tell the RMs to begin listening for connections from the application.

3. The application tells the TM that it is beginning a transaction.

4. The TM issues `xa:start` calls to the RMs, telling them that a new transaction is being started.

5. After issuing one or more commands to the RMs, the application informs the TM that it wants to commit its current transaction.

6. The TM makes an `xa:prepare` call to each of the RMs. Each RM returns a value indicating that it is ready to commit.

7. The TM makes an `xa:commit` call to each of the RMs, indicating that it should commit its work.

The **ax** subservice is used by the RMs to dynamically register with the TM. Once an RM registers with the TM, it is then managed along with all the other RMs that the TM currently knows about. This feature is intended to permit RMs that are infrequently used to be managed only when they are doing actual work, thus avoiding the necessity for them to engage in the transaction protocol when they aren't processing transactions.

7.7.3 The AR Service

The **AR** service is the service an RM provides to allow applications to access its shared resources. The X/Open DTP standard does not specify the form this interface must take. It allows the RMs to provide standard interfaces, such as SQL, and/or custom or proprietary interfaces. The SDTP RM reference implementation currently provides a custom interface built on top of SQL.

A typical call to an **AR** service looks like the following:

```
(wire:remote-value
  rm-wire
  (ar:update *tid*
    (encrypt-string "inv_id" common-key)
    (encrypt-string new-inv-id common-key)
    (encrypt-string " " common-key)
    (encrypt-string " " common-key)))
```

where `rm-wire` is the wire or connection the application has to the resource manager, `ar:update` is the actual RPC to the **AR** service, `*tid*` is the application's thread ID and the rest of the arguments are mapped into an `update` SQL statement.

The SDTP reference implementation of the **AR** service uses encryption to secure the contents of its network communications (as do the other services). The current encryption package uses a Diffie-Hellman key exchange protocol and RC4 as its encryption mechanism.

7.8 Cooperating Law Enforcement Databases

As a test application for the SDTP reference implementation, we built an MLS law enforcement tracking system. This system was inspired by an FBI system called "FOIMS" (Field Office Information Management System), though of course it bears no relationship to the actual working of that system.

Our intent in building this application was to demonstrate multilevel security in a database system that was both distributed and replicated. For this reason we decided to distribute the information for state investigations and replicate the federal information. The motivation for this was that each state would tend to query and update its own information the vast majority of the time and so it would be more efficient to keep that information on local resource managers and allow remote queries; on the other hand, federal information would likely be queried by many states and so it would be more efficient to replicate that information across the resource managers belonging to the states.

An example of a table that contains information classified at varying security levels was the agent table. This table contained a unique ID number for each agent as well as the agent's name. The ID number was needed at all security levels, both to keep track of agent workloads and to ensure that each investigation had a valid agent assigned to it. However, the agent's name was considered

top secret and so it was available only when the user making the query was authenticated at the top secret security level.

Other examples of multilevel tables are the investigation tables. Each investigation was classified at a particular security level; even the fact that an investigation was being conducted on a particular individual might itself be sensitive information. For this reason, investigations that might have, for example, important political implications were classified top secret, while investigations that had resulted in court cases would ordinarily be public knowledge and therefore unclassified.

Our demonstration implementation contained resource managers for two states, each of which also contained a replica of the federal portion of the database. The application was capable of querying, updating, and adding information to the relevant tables.

The SDTP reference architecture enforces the simple security property (*no read-up*) and the $\star$-property (*no write-down*) by using a distributed Security Manager layer. These Security Managers each used a set of single-level databases for storage; the combination of the Security Manager layer and the single-level databases produced a virtual multilevel resource manager.

The Security Managers communicated with the single-level databases by using the native interface provided by the database. The Security Manager would create a set of communication links to the databases and manage them so as to maintain the security constraints. When an operation was requested by a component, the Security Manager would select the set of links that were appropriate for that operation. For example, when a component requested a read operation, the Security Manager would select the links to databases serving the current security level or lower. A write operation would result in only the current security level's link being selected.

The screen dump in Figure 7.6 shows the demonstration instance in operation.

7.9 Related Work

A great deal of work has been devoted to transformational development of software artifacts and to (informal) architecture refinement, but very little has been done that combines both. Other than our own papers on the subject, some of the best examples are Brinksma, Jonsson, and Orava's work on connector refinement [3], work on component refinement [4], and Philipps and Rumpe's work on refinement of information flow architectures [35]. However, work on relating models of connector behavior at different levels of abstraction by Abowd, Allen, and Garlan [1], Luckham, et al. [19], and Moriconi and Qian [25] is closely related.

Similarly, a great deal of work has been devoted to developing verification of transformations, but we are unaware of any work specifically devoted to developing techniques for showing that transformations always introduce faithful interpretations other than our own, with the exception of the sort of very gen-

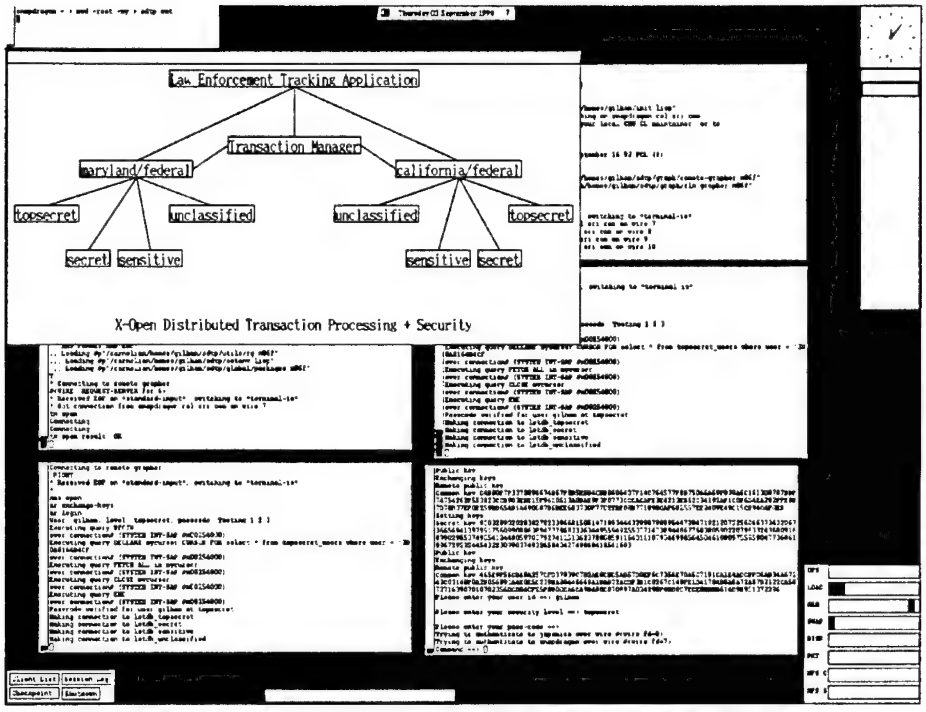

Figure 7.6: SDTP Demonstration Instance in Operation

eral techniques for proving the existence of faithful interpretations found in references such as Turski and Maibaum's book [45].

Generally speaking, the security community has not been successful in formally linking security proofs to actual implementations. A notable exception is Neely and Freeman's verification of a secure gateway [33, 34]. The technique used in Neely and Freemen's case study is not adequate to deal with the SDTP architecture, because it treats only horizontal refinement (i.e., "bubble decomposition") and not vertical refinement (i.e., change in style of representation).

7.10 Chapter Summary

The SDTP case study was performed with several objectives in mind. First, we wanted to determine whether implementation-level SDTP architectural descriptions could be derived for the three implementation approaches defined in our earlier paper [27] by the application of transformations that introduce only faithful interpretations. While we do not have a definitive answer to this question, our experience suggests that such a derivation would be difficult or impossible. Therefore, we attempted to determine whether implementations could be derived using only transformations that always preserve security. The answer to this question is certainly *yes*. But we discovered that defining generally useful transformations that always preserve security can be quite difficult, in some cases. Preservation of security sometimes requires the addition of strong preconditions to the transformation. These strong preconditions can seriously reduce the transformation's generality. We therefore developed an alternative approach allowing transformations that only *sometimes* preserve security to be employed. Rather than building some specific set of sufficient conditions for security preservation into the transformation, we check whether, in each individual application of the transformation, security has been preserved. Our approach to checking is based on a notion we call *proof-carrying architectures* [40]: the same transformation used to implement the architectural description is used to "implement" the security proof, which is carried along with the architectural description. Using a combination of showing that some transformations always introduce faithful interpretations and checking that the others preserve security in the particular case, we succeeded in manually verifying the security of all 12 implementation-level architectural descriptions (four per implementation approach).

A second objective in performing the case study was to begin to determine whether our "rearchitecting via replay" model works as well for global architectural changes (such as introducing security requirements) as it has worked in the 23 small-scale local rearchitecting case studies we have performed based on the X/Open DTP architecture.[4] As we pointed out above, the X/Open DTP derivation was very effectively reused in the defining the SDTP hierarchies. In

[4]The 23 studies are based on making a variety of small changes in system requirements (e.g., adding an RM) and/or implementation infrastructure (e.g., reducing the cost of interprocessor communication).

addition, our techniques for quickly estimating the cost of changes, and conservatively bounding the scope of changes, required to restore consistency in a hierarchy after a change has been introduced continued to work effectively. Therefore, we have begun the process of designing a rearchitecting tool based on this model that we believe will provide very substantial automated support for the process.

Our third objective in performing the case study was to assess the effectiveness of our transformation verification techniques on a larger stock of transformations. Although we wound up abandoning the idea of using only generally valid transformations in the development of the SDTP hierarchies, all the transformations that we believe to be generally valid were successfully verified. So, as mentioned above, we have begun to explore the process of automating these manual verifications using SRI's PVS verification system.

The final principal objective was to determine whether the lowest level architectural descriptions in the SDTP hierarchy were, in fact, implementation-level. In other words, we wanted to confirm our belief that no significant design decisions would have to be made when turning these low-level descriptions into executable code. This is crucial, because the hierarchy effectively links the abstract architecture to the actual implementation only if the gap between the lowest-level description and the code is very small. If the gap is too large, confidence that the "implementation-level" descriptions of SDTP are secure provides only weak evidence that the actual implementation is secure. Our experience was that writing the code from these low-level descriptions is completely straightforward, thanks to formalizing information about implementation language facilities as SADL styles. Although this approach makes the low-level implementation decisions in the SDTP hierarchies implementation-language-dependent, it should hardly be surprising that choice of a programming language for the implementation affects low-level architectural design. Designs appropriate for other choices of programming language can be included by introducing additional branches in the hierarchies (although we have not done so in the case of the SDTP hierarchies we have developed). It should be noted that all these branchings would be at a level of abstraction lower than the X/Open standard, and hence they do not reduce the general utility of an SDTP standard at that same abstraction level. Based on this experience, we are planning to extend our architecture transformation toolset by including a facility for automatic generation of code for the architecture from an implementation-level description in SADL.

So it can be seen that development of the three SDTP hierarchies and the one reference implementation and demonstration instance served as a useful case study in

- transformational development of architectures,
- rearchitecting after adding a new "global" system requirement,
- transformation verification, and
- linking architectural descriptions to code.

Chapter 8

The SDTP Reference Implementation

8.1 The SDTP Architecture

The X/Open DTP standard for distributed transaction processing consists of a protocol specification and a set of services that the components of the system must make available to other system components. SRI's DSA (Dependable System Achitecture) group has used this standard as a testbed for applying our methods for verifying and extending formal descriptions of architectures.

Our most recent project in this area involved extending the X/Open DTP standard to incorporate multilevel security properties. This paper describes a prototype reference implementation that implements this extended standard, along with two applications built using the SDTP system [27].

8.1.1 X/Open DTP

The X/Open DTP standard [47, 49, 48, 46] is intended to standardize the interactions and communications between the components of the 3-tiered client-server model for distributed transaction processing. It allows multiple application programs to share heterogeneous resources provided by multiple resource managers (e.g., database managers, print managers) and allows their work to be coordinated into global transactions.

A version of the X/Open architecture, shown in Figure 8.1, consists of three types of components—one application program (AP), one transaction manager (TM), and one or more resource managers (RMs). The boxes indicate the component interfaces, and the lines indicate the communications between them. The label TX indicates a complex connection and protocol defining communication between any application module and any transaction manager. This connection contains communication channels between functions that initialize and finalize transactions. Communication is always initiated by the application. A series of

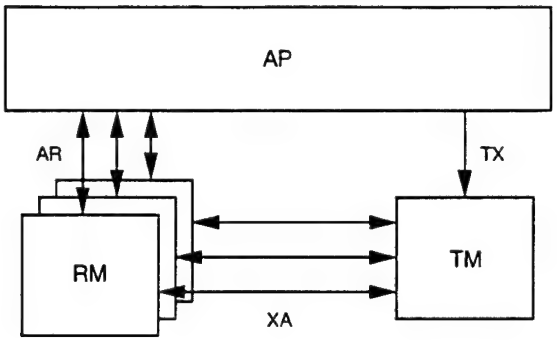

Figure 8.1: X/Open DTP Reference Architecture

calls back and forth continues until communication is completed.

Similar complex connections exist between the application and every resource (the AR connection) and between the transaction manager and every resource manager (the XA connection). The XA connection provides communication for the well-known two-phase commit protocol that ensures the atomicity of transactions. Much of this activity can be concurrent, and many transactions may take place at once.

The X/Open standard talks about these connections in terms of the services (TX, AR, XA) that each component must provide and the interface functions that implement these services. The TX service, also known as the Transaction Demarcation service, must be provided by the TM. The RM provides the AR service, which is the actual data storage interface to the RM. The AP makes use of the TX and AR services. The XA service consists of two subservices, one provided by the TM and the other provided by the RMs. The former, called the AX subservice or AXS, allows RMs to dynamically register and unregister themselves. The latter, called the XA subservice or XAS, exports the procedures the TM uses to coordinate the transactions. Both the TX and XA services are fully specified, albeit informally, while the implementation is allowed to use a custom set of functions for the AR service, allowing implementations to build custom resource managers. Table 8.1 gives a complete listing of the TX and XA services.

8.1.2 Multilevel Security

A standard model of a multilevel security (MLS) policy is the Bell-LaPadula model [16]. Given a set of subjects each with an attached clearance level, and a set of objects each with an attached classification level, the model ensures that information does not flow downward in a security lattice by imposing the following requirements:

Name	Description
TX:BEGIN	Start a new transaction
TX:CLOSE	Close the resource managers
TX:COMMIT	Complete a transaction normally
TX:INFO	Query the TM about the status of a transaction
TX:OPEN	Open the resource managers
TX:ROLLBACK	Abort a transaction
TX:SET-COMMIT-RETURN	Wait for commit completion or just for logging
TX:SET-TRANSACTION-CONTROL	Indicate 'chained' transactions
TX:SET-TRANSACTION-TIMEOUT	Set transaction time limit
AXS:REG	Let the RM register itself with the TM
AXS:UNREG	Let the RM unregister itself
XAS:CLOSE	Tell RM not to listen for connections any more
XAS:COMMIT	Tell RM to commit the transaction
XAS:END	Tell RM to end a transaction
XAS:OPEN	Inform RM that application is opening it
XAS:PREPARE	Ask RM if it can commit the current transaction
XAS:ROLLBACK	Tell RM to abort the transaction
XAS:START	Tell RM to start a transaction for a given application

Table 8.1: TX and XA services

- **The Simple Security Property.** A subject is allowed a read access to an object only if the subject's clearance level is identical to or higher than the object's classification level in the lattice.
- **The * Property.** A subject is allowed a write access to an object only if the subject's clearance level is identical to or lower than the latter's classification level in the lattice.

The MLS policy regulates communication between the application and the resources. As such, it is primarily a property of the architectural structure. That is, it specifies that an application should not be allowed to connect to a resource manager that contains data for which it is not cleared.

8.1.3 Secure DTP

The SDTP implementation is based on a specification generated by formalizing the X/Open DTP specification in SADL [28], the Architecture Description Language used by our group, and then adding the MLS properties to the specification. The resulting specification was refined using provably correct transformations until an implementable specification was produced. The implemented architecture can be seen in Figure 8.2.

The refinement process by which this architecture was produced can be described as starting with communication channels that enforce the multilevel security policy, and then refining them into ordinary communication channels with

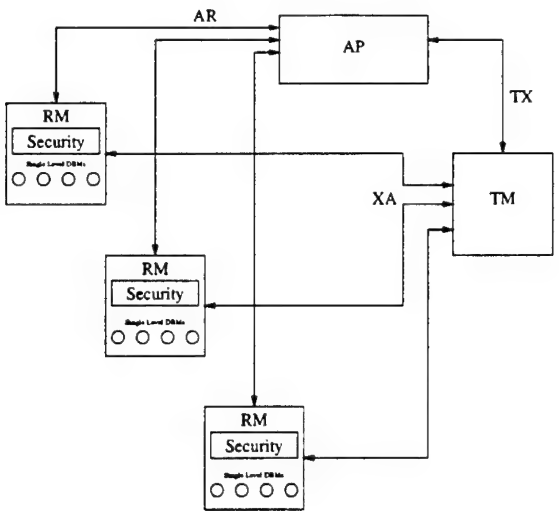

Figure 8.2: X/Open DTP Extended with Multilevel Security

the security policy enforcement implemented by a security manager. Finally, the security manager is distributed as security wrappers around each resource manager. The resource managers themselves are implemented as a collection of single-level database managers. The resulting specification preserves both the desired atomicity properties of X/Open DTP and the security properties mentioned above.

8.2 SDTP System Components

One goal of SDTP was a desire to be able to install it for our client and potentially other interested parties without the need to purchase expensive equipment and software licenses. For this reason, we tried to use freely available software whenever possible. It turned out to be possible to build the system with all major software components being in the public domain or having free redistribution licenses. The following components were important in building SDTP:

- **FreeBSD.** We chose the FreeBSD operating system because we were experienced with it and had found it reliable in the past. It runs on inexpensive Intel-x86 hardware and has a wide variety of freely available software, including database software and Lisp implementations, available through its 'ports' packaging system. In many cases, a desired software package can be installed by finding the appropriate 'ports' directory and issuing a 'make install' command. We found it more stable and less of a moving target than Linux, another freely available UNIX-like operating system.

- **CMU Common Lisp.** The CMU version of Common Lisp [43] is a high-quality Lisp implementation that is in the public domain. We have found its performance to be mostly competitive with, and occasionally better than, commercial implementations. Its compiler also provides informative optimization notes that guide the programmer in making declarations that improve performance.

- **Common Lisp Bignum Facility** To provide secure communications channels, we encrypted the data objects that were sent over them. To do this, we had to implement key exchange and encryption. Common Lisp's bignum facility turned out to be extremely convenient for this purpose. For example, Diffie-Hellman key exchange involved three functions that were each essentially one line of code (omitting declarations), along with a 'modpower' function ![42]:

```
;;; Modpower function from clmath package by Gerald Roylance.
(defun modpower (number exponent modulus)
  (declare (integer number exponent modulus))
  (do ((exp  exponent  (floor exp 2))
       (sqr  number (mod (* sqr sqr) modulus))
       (ans  1))
      ((zerop exp) ans)
    (declare (integer exp sqr ans))
    (if (oddp exp)
        (setq ans (mod (* ans sqr) modulus)))))

(defun compute-secret-key (dh-modulus)
  (declare (integer dh-modulus))
  (random dh-modulus (make-random-state t)))

(defun compute-public-key (base secret-key modulus)
  (declare (integer base secret-key modulus))
  (modpower base secret-key modulus))

(defun compute-common-key (remote-public-key
                           local-secret-key
                           modulus)
  (declare (integer remote-public-key
                    local-secret-key
                    modulus))
  (modpower remote-public-key local-secret-key modulus))
```

While researching implementations we found a version written in the C programming language that took about 90 lines. Much of the C code involved processing a set of arrays, which in effect implemented a bignum capability. As a result, the connection between the C code and the algorithm being implemented was tenuous, while the Lisp code above is

a direct implementation of the algorithm. The bignum facility was also useful in the actual encryption and decryption of data, allowing us to represent intermediate forms of the encrypted data as integers.

- **PostgreSQL.** We used the PostgreSQL[36] relational database manager to provide data storage management for SDTP. The choice of PostgreSQL was based on the fact that we had previous experience with it, it was easy to install and run, and it was licensed so as to allow us to distribute it conveniently.

  PostgreSQL comes with programmatic interfaces for several languages; unfortunately Lisp was not one of them. As a result we had to develop a Lisp interface to PostgreSQL.

  There are at least two ways to implement such an interface. One way is to talk directly to the PostgreSQL back end over the network. This involved creating packets and sending them to the back end. Since at the time we were not completely sure how this worked, we decided to take the second approach, which involved writing a foreign-function interface to the C version of the PosgreSQL client library.

  Like other Common Lisp implementations, CMU Common Lisp has a package that allows Lisp code to talk to code written in C. Interfacing to the PostgreSQL library involved writing a file of interface functions using the CMUCL Foreign Function Interface. It turned out to be easy to write the code. There were a few opaque data structures, some enumerations for status returns, and a series of functions. The following is a sample of the code

```
;; Opaque data structures.
(def-alien-type postgres-connection system-area-pointer)
(def-alien-type postgres-result system-area-pointer)

;; Status enumerations.
(def-alien-type connection-status (enum nil
                                        :connection-ok
                                        :connection-bad))

(def-alien-type exec-status (enum nil
                                  (:empty-query 0)
                                  :command-ok
                                  :tuples-ok
                                  :copy-out
                                  :copy-in
                                  :bad-response
                                  :nonfatal-error
                                  :fatal-error))
```

```
;; Some of the actual library interface functions.
(declaim (inline PQsetdbLogin))
(def-alien-routine "PQsetdbLogin" postgres-connection
  (pghost c-string)
  (pgport c-string)
  (pgoptions c-string)
  (pgtty c-string)
  (dbname c-string)
  (login c-string)
  (passwd c-string))

(declaim (inline PQexec))
(def-alien-routine "PQexec" postgres-result
  (connection postgres-connection)
  (command c-string))

(declaim (inline PQresultStatus))
(def-alien-routine "PQresultStatus" exec-status
  (result postgres-result))

(declaim (inline PQntuples))
(def-alien-routine "PQntuples" int
  (result postgres-result))

(declaim (inline PQnfields))
(def-alien-routine "PQnfields" int
  (result postgres-result))

;; etc.
```

A couple of problems arose from using this method. The main problem was due to an interaction between FreeBSD's object format and CMUCL. CMUCL is currently linked statically under FreeBSD, which means it cannot use shared binary objects. The PostgreSQL C library is built using shared objects, for both its shared and static versions. (We think this is a bug but were unable to convince the maintainers to change it.) At any rate, we were forced to create by hand a library using static objects.

A similar problem involved the process of actually loading the library into the running Lisp. Since CMUCL running under FreeBSD uses an old technique of runtime linking, rather than the newer technique of dynamic loading, the C code library can be loaded only once per session; multiple loadings result in multiple definition errors.

- **Remote Procedure Call.** SDTP is implemented as a set of components and communication facilities. Our desire was to try to reflect current practice in implementing a DTP system. Thus, communication between

the components is done using a Lisp-based Remote Procedure Call (RPC) mechanism.

The components of SDTP employ a Lisp-based RPC mechanism for all communication except the communication with the database managers (which is done through the programmatic interface library discussed above). The RPC mechanism used is the Remote package provided by CMU Lisp. It is built on a lower-level socket based package called the Wire package. The Remote package allows calls like the following:

```
(wire::remote-value rm-wire (xas:open tmid rmid flags))
```

where `rm-wire` is a handle on the communication link over which the call is being made. This call causes the

```
(xas:open tmid rmid flags)
```

procedure call to be evaluated in the remote Lisp process referenced by the `rm-wire` handle.

The Remote package has several limitations. First, it is limited in the data objects it can actually send between the communicating processes. It lacks a fully general external data representation (XDR) mechanism. On the other hand, it allows a remote Lisp process to refer to a local data structure by passing a token called a remote object. The remote process can, without actually modifying the data object, effectively pass it back as a return value or pass it as a parameter to a call-back procedure it invokes in the local process. Communicationds channels created by the Remote package are bidirectional; a remote Lisp process can invoke procedures in the local process as well as the other way around. If a process needs to actually send data objects that are not supported by the Remote package, it must convert them to a string format and send them as strings, whereupon the remote process must convert them back into data objects. In practice, limitations of data object types did not arise in this system.

Another limitation was the fact that the Remote package provided no access control. A process making remote procedure calls can invoke any procedure in the remote Lisp process, if it knows the package and procedure name. This creates a security hole; at the very least a malicious client could invoke a denial-of-service attack by remotely calling the `(quit)` procedure. It would be fairly straightforward to create a version of the Remote package that required each process to register procedures that it would allow to be invoked remotely; such an extension was not made in this implementation but will probably be done in the future.

- **Graphical User Interface Toolkit.** At first, SDTP did not have a GUI. In the form we originally demonstrated, it had a graphical display to show

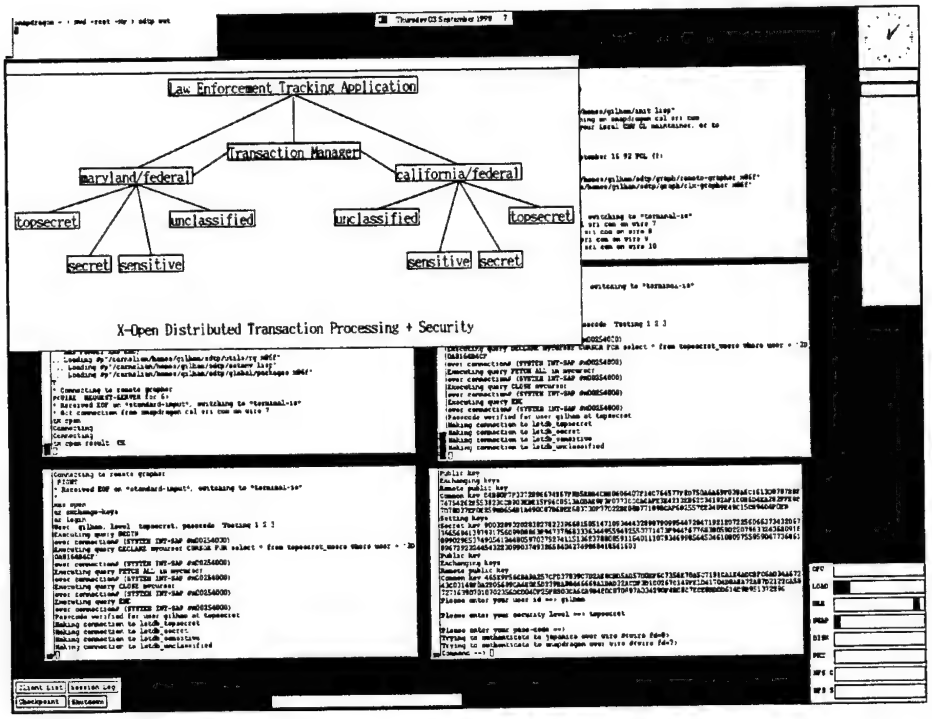

Figure 8.3: SDTP with Component Interaction Graph

the interactions between the components (see Figure 8.3). To make the demonstration application more usable, we decided to give it a GUI. All the GUI work in SDTP, including the component-interaction graph, was done using the Garnet [9] toolkit.

The Garnet toolkit, besides being freely distributable, had good performance and a relatively small footprint in CMUCL. An x86 Lisp image with Garnet is about 18 MB, while the image without Garnet is about 14 MB (CMUCL's images are considerably larger than current commercial versions). This compares to a size of around 28 MB for a CLOS-based toolkit such as XIT/CLUE [14].

Originally we were concerned about the fact that Garnet did not use CLOS as its object system; instead it uses KR, a relatively simple prototype-instance object system with constraints. However, with experience we came to feel that KR's approach was a good match for GUI development, and its (relatively) small size and low overhead served us well when we installed the system on laptops and other resource-limited machines. At one

point we had several Lisp processes, including a Garnet process, running on a 32 MB 200 MHz Pentium machine. Performance, while not sterling, was adequate. CMUCL uses the PCL version of CLOS. Experience with the XIT toolkit showed that it caused PCL to spend a lot of time doing runtime compilation, especially when it loaded files, while Garnet did little or no runtime compilation.

Among Garnet's important features that we used heavily were objects known as 'interactors' that deal with various types of user input. Interactors can be attached to different graphical objects, thus almost completely decoupling the look of the object from the way the object responds to user interaction. As a result of the use of interactor objects, Garnet allows a programming style that minimizes the use of callbacks. Instead of an event-driven model where the programmer writes procedures that are given control in response to user actions, the programmer can use a more functional approach. We ended up in effect calling the GUI as a function that returns the results of the user's activity. The messy event handling and synchronization is kept neatly behind the scenes by the interactor mechanism.

Garnet provides two toolkits with different appearances. We used both; for the component-interaction graph we used Garnet's native toolkit, while for the application GUIs we used the second toolkit that provides a Motif-like appearance.

8.3 Applications

We built two applications on top of SDTP. The first is a demonstration application used to show that SDTP works and has the desired security properties. The second is intended as the genesis of a real-world application that would serve as an intelligent post-processor to a computer intrusion-detection system. Figure 8.4 illustrates the GUI style of both applications.

8.3.1 Cooperating Law Enforcement Databases

As a test application we built a multilevel-secure law-enforcement-tracking database, LET-DB. This was inspired by an FBI system called "FOIMS" (Field Office Information Management System), though of course it bears no relationship to the actual working of that system.

Our intent in building this application was to demonstrate multilevel security in a database system that was both distributed and replicated. For this reason we decided to distribute the information for state investigations and replicate the federal information. The motivation for this was that each state would tend to query and update its own information the vast majority of the time and so it would be more efficient to keep that information on local resource managers and allow remote queries; on the other hand, federal information would probably

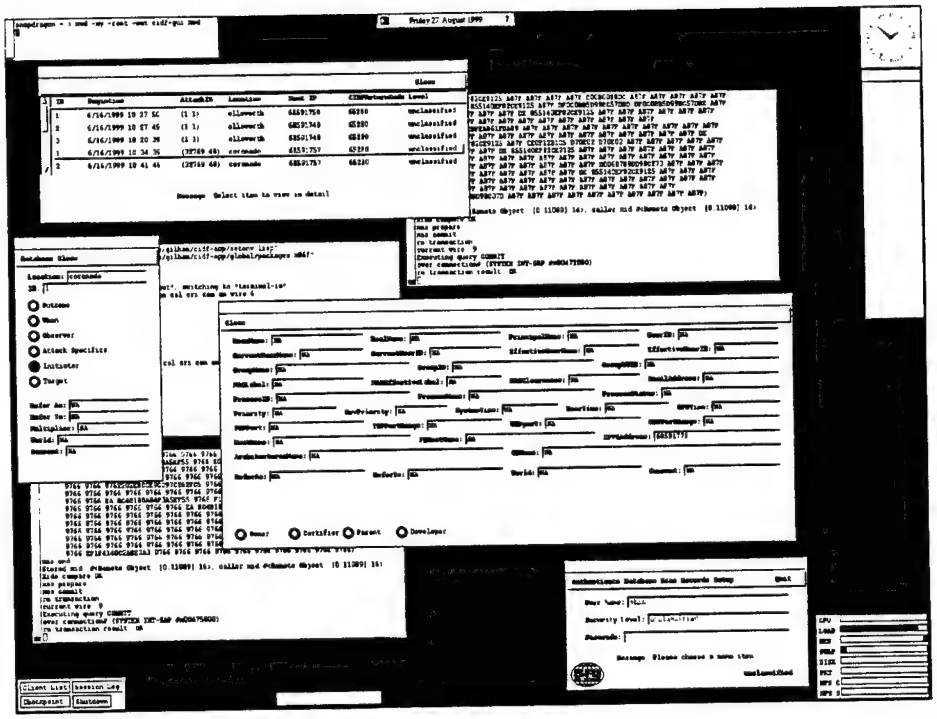

Figure 8.4: SDTP Application GUI

be queried by many states and so it would be more efficient to replicate that information across the resource managers belonging to the states.

An example of a table that depended on multilevel security was the agent table. This table contained a unique ID number for each agent as well as the agent's name. The ID number was needed at all security levels, both to keep track of agent workloads and to ensure that each investigation had a valid agent assigned to it. However, the agent's name was considered top secret and so it was available only when the user making the query was authenticated at the top secret security level.

Other examples of multilevel tables were the investigation tables themselves. Each investigation was classified at a particular security level; even the fact that an investigation was being conducted on a particular individual might itself be sensitive information. For this reason, investigations that might have, for example, important political implications were classified top secret, while investigations that had resulted in court cases would ordinarily be public knowledge and therefore unclassified.

Our demonstration implementation contained resource managers for two states, each of which also contained a replica of the federal portion of the database. The application was capable of querying, updating, and adding information to the relevant tables.

8.3.2 Intrusion Detection Application

The second application built on SDTP is the Intrusion Detection Application. One of the authors of this paper, David Shih, was hired as a summer intern to write this application. He has completed his first year as a Computer Science major at the University of California at Berkeley. His most significant programming experience was his first-semester computer science programming course, taught using Scheme. Thus he, like the other author, was writing his first major Lisp program.

The Intrusion Detection Application turned out to be larger and more complex than the Law Enforcement Tracking application. Like the LET-DB application, the IRDB (Intrusion Recording Database) also manages multilevel secure databases, this time containing computer intrusion data. This application receives, from one or more secure sources, CISL (Common Intrusion Specification Language) data. CISL is a proposed standard for representing and exchanging computer intrusion data. A complete draft on CISL and its syntax can be found at

`http://gost.isi.edu/projects/crisis/cidf/cisl_current.txt.`

Conveniently, CISL uses an s-expression format for its external representation.

In our case, the application receives CISL data describing intrusion attempts against machines located at various military installations. (For example, our current demonstration pretends to receive data from two, Ellsworth AFB and NAB Coronado). The application currently reads CISL data, classifies each entry, and inserts the entry into the proper database. A user can then query

the database, retrieve CISL entries, and look through an entry in detail in a more user-friendly environment. Eventually the application will perform intelligent pattern-matching across the database entries to check for clues that would suggest a serious breach attempt against the network

Represented as s-expressions in a tree structure, CISL attack entries express specific information about intrusion attempts by describing the verb/action (in this case, Attack) with a sublayer of role and adverb entries. These describe the "players" involved and attributes of the verb, respectively. Beneath role entries are attribute entries describing role attributes such as the owner or developer of that particular player. All these entries—verb, role, adverb, and attribute—contain what are known as atomic entries. An atomic entry contains the actual values which describe the entry within which it is enclosed. So a sample generic structure for a CISL entry could look something like

```
(Attack                                ; verb SID (semantic identifier)
  (World Redhat-5.1)                   ; atomic SID and value
  (Outcome                             ; adverb SID
    (atomSID2    value2)
    (atomSID3    value3))
  (Observer                            ; role SID
    (atomSID4    value4)
    (atomSID5    value5)
    (Owner                             ; attribute SID
      (atomSID6 value6))))
```

Currently, to store data, a batch-mode data-storage application simulates the remote data streams that we plan to use. Since the CISL data we read has no provision for classification of records, the batch-mode program reads the input and classifies the data. To do this, it performs a tree search for the atomic field containing the IP of the targeted machine. From the IP, it determines the record's security classification and location. The intuition behind this is that there may be sensitive information exposed by an intrusion attempt—such as an unaddressed vulnerability of a classified machine, or even the existence of said machine—that users below a certain security level should not have access to. By classifying records by target IPs, records about these machines can be hidden away at the proper level. Eventually, however, instead of receiving data from a single mixed data stream and having the application organize the data into different classification levels, the application will receive information from pre-classified data streams, allowing the application to simply tag the data with the security level associated with the stream from which it was received, and insert it into the proper database.

As mentioned above, input into IRDB can be given only through a secure data stream containing raw CISL data. The application then has to parse the CISL s-expressions into string entries that can be stored into the PostgreSQL database. Because CISL was designed to be extensible, insertion of CISL s-expressions into a relational database becomes a chore because there is no guaranteed uniformity between one CISL entry and the next. Atomic fields and

sub-rows that existed in one CISL construct may not exist in another. As a result, the application has to make sure that existing atomic fields that were not reported in a CISL entry are filled in with some kind of no-data identifier and that role entries and adverb entries to the attack entry have a similar procedure applied to them and to their attribute entries. These new "completed" rows, which now all contain the same of number entries, are then inserted into their respective tables.

Consequently, because of the layered makeup and the potentially large size of a CISL entry, it seemed more sensible to break up the large CISL entry into its verb, role and adverb entries and insert each of those entries into its own separate table rather than create a table with more than 200 columns. To maintain uniformity across the multiple tables, the application tags each subentry with a unique ID number. In dealing with the attribute entries, each role entry that allows for attribute entries has columns in its table reserved for each attribute. The application then simply inserts attribute entries in their s-expression form as a whole into the column within the role table to which they belong. Since all attribute entries do actually have the same number and type of fields, it didn't seem necessary to create sixteen extra tables just to store them. Instead, we use the process described above and employ a kind of lazy procedure to format the attribute entry from s-expression to string when the user actually asks that a particular attribute item be shown in detail after the query results are displayed during lookup.

8.4 Chapter Summary

Both authors of the applications were writing their first major Lisp program. One has more than fifteen years of programming experience in everything from assembly language and microprocessor BASIC to C; the other was writing his first major program in any language. Both programmers were pleased with Lisp as part of a development infrastructure.

A major feature of our development experience was the interactive programming environment. We used the Ilisp mode for Emacs, which gave convenient access to the interactive features of Lisp.

This took getting used to. One of us, used to the C batch-mode development environment, found himself constantly killing off the Lisp process and restarting it every time he made a change. Eventually, it dawned on him that it wasn't necessary to do this. It was a revelation when he modified the text of a function and, using the Ilisp `(compile-defun-and-go)` command, added it to the currently running system and saw the change take immediate effect. This was especially helpful because, upon restarting the system, it often took a minute or so to get back to the point where the change could be tested.

Another instance where the interactivity of Lisp was valuable was at the conference where the system was first demonstrated. The demonstrator made a typo that caught a bug in the application. This resulted in the application entering the Lisp debugger with a segmentation violation error. (This error

indicates that the process has made an invalid memory access, for example, to an area of memory that is not currently part of its valid address space.) The demonstrator quit the debugger, getting back to the Lisp toplevel, and then re-entered the main loop of the application and picked up where he left off. The total time consumed by the crash was about ten seconds, and it appeared that the person watching the demo didn't actually know that the system had crashed. This is in contrast to the chaos that would have resulted if an equivalent C program had gotten a segmentation violation.

The component-visualization grapher turned out to be an extremely effective part of the demo, helping people understand the interactions between the components of the system. It gave a real-time view of what was going on in the system and made it clear how the multilevel security aspects operated. This grapher was written using Garnet and was easy to implement and test.

We have already described the major enhancements and additions we made to the system—adding a GUI to the original application and writing a second application. As we mentioned above, the task of adding a GUI to a command-line-oriented application was simplified by the programming style allowed by the Garnet toolkit. Instead of having to turn the program flow inside out to conform to the event-driven model of typical X-based GUI toolkits, we were able to use a functional approach that retained the original program flow. Below is part of the main loop of the application. The main change from the original text-based UI is the use of the calls to the SG (SDTP-GUI) package instead of using text-based alternatives.

```
(defun do-application ()
(loop
 (let* ((user (if *resource-managers*
                  (rm-data-user (car *resource-managers*))
                nil))
        (command (sg:application-interface
                     user
                     (if (minusp *current-level*)
                         ""
                       (level-number-to-name *current-level*)))))
   (sg:toplevel-message "Please choose a menu item.")
   (case command
     (:login
      (setup-tm-connection)
      (setup-rms)
      (login))
     (:quit                              ; Logout and close GUI.
      (shutdown)
      (sg:force-quit)
      (return-from do-application nil))

      ;; etc
```

In general, we found that the use of Lisp made programming the system a pleasant experience. We were able to use a wide variety of solutions to the various problems the system posed without fear that debugging new approaches would be a tedious, time-consuming process. Both developers learned new programming techniques as a result of building this system. Most important, our superiors and clients were impressed with the results of what we produced.

Bibliography

[1] G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. Technical Report CMU-CS-95-111, School of Computer Science, Carnegie Mellon University, January 1995.

[2] J. Baldwin. Definable second-order quantifiers. In J. Barwise and S. Feferman, editors, *Model-Theoretic Logics*. Springer-Verlag, 1985.

[3] E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of TAPSOFT '91*, pages 71–80. Springer-Verlag, 1991.

[4] M. Broy. Compositional refinement of interactive systems. Technical Report No. 89, Digital Systems Research Center, Palo Alto, July 1992.

[5] M. Broy. Compositional refinement of interactive systems. To appear in *JACM*. Available on the Web at `http://www.jacm.org/papers/jacm1061.ps`. 1997.

[6] C. C. Chang and H. J. Keisler. *Model Theory*. North-Holland, 1973.

[7] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, second edition, 1994.

[8] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[9] Brad A. Meyers, et al. The Garnet Reference Manuals. Technical Report CMU-CS-90-117-R5, School of Computer Science, Carnegie Mellon University, December 1994.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '94)*, pages 179–185. ACM Press, 1994.

[11] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architectural description interchange language. In *Proceedings of CASCON '97*, November 1997. Available at

http://www.cs.cmu.edu/afs/cs/project/abel/www/acme-web/v3.0/white-paper-v3.0/white-paper.html.

[12] D. Garlan and D. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[13] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, April 1982.

[14] Jürgen Herczeg, Hubertus Hohl, and Matthias Ressel. XIT — *The X User Interface Toolkit, Programming and Reference Manual.* Research Group DRUID, Department of Computer Science, University of Stuttgart, February 1995.

[15] W. Hodges. *Model Theory.* Cambridge, 1993.

[16] C.E. Landwehr. Formal models for computer security. *ACM Computing Survey*, 13(3):247–278, September 1981.

[17] L. J. LaPadula and D. E. Bell. MITRE technical report 2547, Volume II. *Journal of Computer Security*, 4(2,3):239–263, 1996.

[18] E. J. Lemmon. *Beginning Logic.* Chapman and Hall, second edition, 1987.

[19] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[20] B. Mates. *Elementary Logic.* Oxford University Press, second edition, 1972.

[21] D. McCullough. A hookup theorem for multilevel security. *IEEE Transactions on Software Engineering*, 16(6):563–568, June 1990.

[22] J. McLean. The specification and modeling of computer security. *IEEE Computer*, 23(1):9–16, January 1990.

[23] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 79–93, Oakland, California, May 1994.

[24] J. D. Monk. *Mathematical Logic.* Springer-Verlag, 1976.

[25] M. Moriconi and X. Qian. Correctness and composition of software architectures. In *Proceedings 2nd ACM Symposium on Foundations of Software Engineering (SIGSOFT '94)*, pages 164–174. ACM Press, 1994.

[26] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995. Available at http://www.sdl.sri.com/dsa/publis/tse95.ps.gz.

[27] M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure software architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 84–93, May 1997. Available at `http://www.sdl.sri.com/dsa/publis/sp97.ps.gz`.

[28] M. Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, March 1997. Available at `http://www.sdl.sri.com/dsa/publis/sadl-intro.ps.gz`.

[29] C. Morris and C. Ferguson. How architecture wins technology wars. *Harvard Business Review*, pages 86–96, March–April 1993.

[30] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. Submitted to PLDI '98. Available at `http://www.cs.cmu.edu/~necula/pldi98.ps.gz`.

[31] George C. Necula and Peter Lee. Proof-carrying code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, November 1996. Available at `http://www.cs.cmu.edu/~necula/tr96-165.ps.gz`.

[32] George C. Necula and Peter Lee. Efficient representation and validation of logical proofs. Technical Report CMU-CS-97-172, School of Computer Science, Carnegie Mellon University, October 1997. Available at `http://www.cs.cmu.edu/~necula/tr97-172.ps.gz`.

[33] R. B. Neely and J. W. Freeman. Structuring systems for formal verification. In *Proccedings of the IEEE Symposium on Security and Privacy*, pages 2–13, April 1985.

[34] R. B. Neely, J. W. Freeman, and M. D. Krenzin. Achieving understandable results in a formal design verification. In *Proceedings of the Computer Security Workshop II*, pages 115–124, June 1989.

[35] J. Philipps and B. Rumpe. Refinement of information flow architectures. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, pages 203–212, November 1997. Available at `http://www4.informatik.tu-muenchen.de/papers/icfem_rumpe_1997_Publication.html`.

[36] The PostgreSQL Development Team. *PostgreSQL User's Guide*, 1999.

[37] M. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, Fairfax, Virginia, November 1994.

[38] R. A. Riemenschneider. How to prove faithfulness of elementary theory interpretations. SRI CSL Dependable System Architecture Group, Working Paper DSA-97-01. Available at `http://www.sdl.sri.com/dsa/publis/proof.ps.gz`., October 1997.

[39] R. A. Riemenschneider. A simplified method for establishing the correctness of architectural refinements. SRI CSL Dependable System Architecture Group, Working Paper DSA-97-02. Available at `http://www.sdl.sri.com/dsa/publis/simplified.ps.gz`., November 1997.

[40] R. A. Riemenschneider. Checking the correctness of architectural transformation steps via proof-carrying architectures. Available at `http://www.sdl.sri.com/dsa/publis/checking.ps.gz`., April 1998.

[41] R. A. Riemenschneider. Correct transformation rules for incremental development of architecture hierarchies. SRI CSL Dependable System Architecture Group, Working Paper DSA-98-01. Available at `http://www.sdl.sri.com/dsa/publis/incremental.ps.gz`., February 1998.

[42] Gerald Roylance. Some scientific subroutines in LISP. Technical Report AIM-774, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, September 1984.

[43] School of Computer Science, Carnegie Mellon University. *CMU Common Lisp User's Manual*, July 1992.

[44] J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.

[45] W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.

[46] X/Open Company, Apex Plaza, Forbury Road, Reading, Berkshire RG1 1AX, U.K. *Distributed Transaction Processing: Reference Model, version 2*, November 1993.

[47] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The XA Specification*, June 1991.

[48] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The Peer-to-Peer Specification*, December 1992.

[49] X/Open Company Ltd., Apex Plaza, Forbury Road, Reading, Berkshire RGI 1AX, U.K. *Distributed Transaction Processing: The TX (Transaction Demarcation) Specification*, November 1992.

*U.S. GOVERNMENT PRINTING OFFICE: 2000-510-079-10004

DISTRIBUTION	# of Copies
MICHAEL P. NASSIF AFRL/IFSB 525 BROOKS RD ROME NY 13441-4505	5
SRI INTERNATIONAL 333 RAVENSWOOD AVE MENLO PARK CA 94025-3493	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	1
DARPA/ITO 3701 N. FAIRFAX DRIVE ARLINGTON VA 22203-1714	2

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.